WIND™

VXWORKS 7®

PROGRAMMER'S GUIDE

Edition 5

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 7
Programmer's Guide

Edition 5
9 Jul 15

Contents

1
Kernel Applications

1.1 About Kernel Applications

VxWorks kernel applications execute in the same mode and memory space as the kernel itself. In this respect they are unlike applications used with other operating systems, such as UNIX and Linux; and also unlike VxWorks real-time process (RTP) applications.

Kernel applications can either be interactively downloaded and run on a VxWorks target system, or linked with the operating system image and (optionally) configured to execute automatically at boot time.

VxWorks applications that execute in the kernel are created as relocatable object modules. They can be referred to most specifically as *kernel-based application modules*, but it is often convenient to refer to them simply as *kernel application modules* or *kernel applications*, as is done in this guide. (They are sometimes referred to as *DKMs*, after the abbreviation for the Workbench project type Downloadable Kernel Module, even when they are linked with the kernel.) Kernel applications should not be confused with applications that execute in user mode as real-time processes (RTPs).

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

Kernel application modules can be either:

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

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

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

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

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

WARNING: If you wish to port a kernel application to a user-mode application that executes as a real-time process (RTP), you must ensure that it meets the code requirements of an RTP application and is compiled as such. You must also ensure that VxWorks is configured with support for RTPs. For more information, see *22. Kernel to RTP Application Migration*.

Kernel Applications and Kernel Component Requirements

VxWorks is a highly configurable operating system. When kernel application modules 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, useful for application code to check for errors indicating that kernel facilities are not available (that is, check the return values of API calls) and to respond appropriately.

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

Binary Compatibility

⚠ **CAUTION:** Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

⚠ **CAUTION:** The kernel object-module loader rejects a module if it is not compatible with the VxWorks system, prints an error message, and sets **errno** (to **S_loadElfLib_HDR_READ** when the architectures are different or to **S_loadLib_INCOMPATIBLE_MODULE** when other features are different).

Comparing Kernel Applications with RTP Applications

The primary difference between kernel applications and RTP applications is that RTP applications execute in separate, protected memory environments, while kernel applications do not. There are also differences in the features available in each environment, as well as differences in performance, footprint, and memory use.

VxWorks kernel applications execute in kernel mode in the kernel memory space. They can be linked with the kernel or downloaded into it at runtime (as downloadable kernel modules—DKMs). Because kernel applications execute in the kernel, the system cannot protect the kernel from misbehavior on the part of applications, nor applications from one another. At runtime, VxWorks system tasks (such as **tExcTask**) and application tasks are distinguished only by their priority and the functionality of their code.

RTP applications execute as real-time processes in user mode, each in a their own protected memory space, which is separate from that of other RTP applications and from the kernel. RTP applications should be used when this protection is a key consideration for the system. RTP applications are built and stored separately from the kernel (on the host during development and on the target for deployed systems). VxWorks RTP applications are similar to applications for other operating systems such as UNIX and Linux, except that VxWorks real-time processes are designed for real-time systems (for more information in this regard, see *2. Real-Time Processes: RTPs*).

The kernel application environment and the RTP application environment have many of the same features, but also differ in what is available in each. For example, drivers can only be installed in the kernel, and PSE52 conformance can only be achieved in the RTP environment.

While there is no inherent difference in the speed of code executing in the kernel environment or the RTP environment, it can be affected by code optimization (although less so in the RTP environment) and the use of system calls (which are specific to the RTP environment). RTP applications that make notable use of system calls—which is generally the case—are slower than comparable kernel applications. In addition, VSB projects can be used to remove RTP-handling code from various VxWorks libraries (using the **Real-Time Process** option), which improves the performance of those libraries and reduces their size.

In addition, a VxWorks image configured with RTP support components will be slightly larger than one that is not. And the overall memory requirements for a system that employs RTP applications is generally larger than one that uses comparable kernel applications because RTP applications include user-side libraries (which "duplicate" those already included in the kernel).

1.2 C and C++ Libraries

Both VxWorks native C libraries, and Dinkum C and C++ libraries, are provided for VxWorks application development.

Use of VxWorks Native Libraries and Dinkum Libraries

VxWorks native libraries are used for C kernel application development, and Dinkum libraries are used in all other cases.

Table 1-1 **C and C++ Libraries**

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

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

For more information about these libraries, see the VxWorks and Dinkum API references. For more information about C++ facilities, see *5. C++ Development*.

1.3 Kernel Application Structure

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

Application Entry Point

The entry-point function performs any data initialization that is required, and starts all the tasks that the running application uses.

NOTE: If your kernel application includes a **main( )** function, do not assume that it will start automatically. Kernel application modules that are downloaded or simply stored in the system image must be started interactively (or be started by another application that is already running). The operating system can also be configured to start applications automatically at boot time (see a*1.8 Using userAppInit( ) to Run Applications Automatically*, p.11).

For example, a kernel application might have a function named like **myAppStartUp( )**, which could look something like this:

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

For information about VxWorks tasks and multitasking, see *6. Multitasking*. For information about working with C++ see *5. C++ Development*.

Automatic Invocation of Initialization and Termination Functions

If you are going to download your kernel application module to VxWorks at runtime, you can have the loader start it automatically by implementing a **_VxModuleAutoInit( )** function as the entry point, and using the **LOAD_CALL_INIT_AUTO** option with **loadModuleAt( )**. The **LOAD_CALL_INIT_MANUAL** option is also provided so that the module can be loaded without calling the initialization function automatically.

Similarly, you can implement a **_VxModuleAutoTerm( )** function that is automatically invoked when the module is unloaded using the **unldByModuleId( )** function with the **UNLD_CALL_TERM_AUTO** option.

For more information, see the **loadLib** and **unldLib** API reference entries.

1.4 VxWorks Header Files

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

VxWorks System Header Files

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

VxWorks system header files are copied to the VSB project directory, below *vsbProjectDir*/**krnl/h**.

VxWorks Header File: vxWorks.h

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

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

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 library. The *tty* header file **tyLib.h** uses definitions that are supplied by the ring buffer header file **rngLib.h**.

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

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

VxWorks Private Header Files

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

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

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

Internal details are also hidden with *private* header files, which are located in directories named **private**. The names of the private header files have the same root as the public ones, but with a **P** added. For example, the private header file for **semLib** is **semLibP.h**.

Header Files and extern "C" Statements

Do not include system header files within **extern "C"** statement brackets. All VxWorks system header files are already configured for use with C and C++ compilers. You may invalidate declarations in the VxWorks headers if you force a C linkage on the entire header file's content. The following is incorrect:

```
#ifdef _cplusplus
extern "C" {
#endif

#include <stdio.h>
```

But the following is correct:

```
#include <stdio.h>

#ifdef _cplusplus
extern "C" {
#endif
```

In addition, do not include a VxWorks header file within an **extern "C"** statement in C++ files. The following is incorrect:

```
extern "C" {
#include <stdio.h>
}
```

But this example is correct:

```
#include <stdio.h>
```

1.5 Static Instantiation of Kernel Objects

VxWorks kernel objects—such as tasks and semaphores—can be instantiated statically as well as dynamically. Static instantiation provides advantages in terms of performance and determinism. Special VxWorks C macros are provided for static instantiation.

Static instantiation of a kernel object means that the object is declared as a compile-time variable (using a special VxWorks macro), which is usually global in scope. The compiler therefore allocates storage for the object in the application, and no run-time memory allocation is required. The object is therefore immediately available for initialization at startup.

In contrast to static instantiation, dynamic instantiation of kernel objects involves run-time allocation of system memory and then initialization of the object before it can be used. Object deletion, in turn, involves invalidation of the object, and then return of the memory to the system. Object creation and deletion are both dependent on dynamic memory allocation, usually with the **malloc( )** and **free( )** functions.

Applications using dynamic instantiation must therefore account for the possibility that the system may run out of memory and that it may not be able to create the object (and resort to some suitable error recovery process or abort). In addition, dynamic allocation is a relatively slow operation that may block the calling task, and make system performance less deterministic.

Also in contrast to static instantiation, dynamic instantiation can be accomplished with a single function call (for example, **taskSpawn( )**, **sem*X*Create( )**, and so on).

The code examples that follow illustrate the difference between coding dynamic and static instantiation (but do not use any VxWorks object instantiation macros).

Dynamic Instantiation

```
struct my_object * pMyObj;
    ...
pMyObj = (struct my_object *) malloc (sizeof (struct my_object));
if (pMyObj != NULL)
    {
```

```
    fooObjectInit (pMyOjb);
    return (OK);
    }
else
    {
    /* failure path */
    return (ERROR);
    }
```

Static Instantiation

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

Kernel Objects That can be Instantiated Statically

The following kernel objects can be instantiated statically:

- kernel tasks
- semaphores
- message queues
- watchdog timers

For detailed information, see *Static instantiation of Kernel Tasks*, p. 91, *Static Instantiation of Kernel Semaphores*, p. 118, *Static Instantiation of Kernel Message Queues*, p. 133, and *Static Instantiation of Watchdog Timers*, p. 174.

Static Instantiation and Code Size

Compile-time declaration of objects does not occupy any space in an executable file, in a VxWorks image, or in storage media (such as flash memory). If an object is declared at compile time but not initialized, the compiler places it in the un-initialized data (bss) section. Un-initialized data is required by the ANSI C standard to have a value of zero.

While un-initialized data contributes to the run-time memory footprint, so does dynamic allocation. In either case, the memory footprint would be the same.

Advantages of Static Instantiation

Static instantiation of kernel objects provides several advantages over dynamic instantiation:

- Static instantiation of objects is faster and more deterministic.
- Application logic is simpler because it does not have to account for the possibility that dynamic allocation may fail.
- Static declaration of objects cannot fail, unless the program itself is too large to fit into system memory.

Applications and Static Instantiation

Static instantiation of kernel objects provides notable advantages to real-time applications, and most applications can make use of static instantiation to some degree. Most applications require that some objects be available for the lifetime of their execution, and that they never be deleted. These objects can therefore be statically instantiated. Using static instantiation whenever possible makes applications more robust, deterministic, and fast.

NOTE: Static instantiation should only be used with kernel applications. It is not designed for use with RTP (user-mode) applications.

Scope Of Static Declarations

Kernel objects are usually declared as global variables because the objects IDs are usually used for inter-task communication and synchronization. However, they need not be global. An object declaration can also be done at function scope provided the object stays in a valid scope for the duration of its use.

Caveat With Regard to Macro Use

In order to ensure proper macro expansion, the task instantiation macros must use a backslash character if the declaration or call that they implement is longer than one line. For example, the following ensures that the static instantiation of a task with the **VX_TASK_INSTANTIATE** macro will be handled properly by the compiler:

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

Macro use is described in detail in *Static instantiation of Kernel Tasks*, p. 91, *Static Instantiation of Kernel Semaphores*, p. 118, *Static Instantiation of Kernel Message Queues*, p. 133, and *Static Instantiation of Watchdog Timers*, p. 174.

1.6 Boot-Time Hook Function Facility

VxWorks hook function stubs that are called at specific points in the boot process can be used to add your own boot-time functionality or test code. The stubs allow you to add code without modifying the boot loader or operating system code itself, or without creating custom components for your own code.

You can add code that performs time-critical operations before the operating system is fully up and running, or code that overrides default features of the system. For example, you can use this hook function functionality to do the following:

- Provide early feedback that the device is booting (print a message to an LCD display).
- Reset a watchdog timer during the boot process.
- Call custom diagnostic or test code.
- Start applications that do not require networking earlier in the initialization process.
- Read a GPIO for hardware version or test device status.
- Configure an FPGA early.

The boot-time hook function stubs are enabled by including stub-specific components in your VxWorks configuration. The stubs must be called from the **usrAppInit.c** file (in your VIP project).

For information about using **usrAppInit.c** to automatically executing code after VxWorks has booted, see *1.8 Using userAppInit( ) to Run Applications Automatically*, p.11.

1.7 Boot-Time Hook Function Stubs and Components

The boot-time hook function stubs are enabled by including stub-specific components in your VxWorks configuration.

Function Stubs and Components

Table 1-2 **Boot-Time Hook Function Stubs and Components**

Hook Function Stub	Description	Component Required
usrPreKernelAppInit( )	Called at the beginning of **usrInit( )**, right after **cacheLibInit( )**. That is, right after the board boots and before kernel initialization starts.	**INCLUDE_USER_PRE_KERNEL_APPL_INIT**
usrPostKernelAppInit( )	Called right after **usrKernelCreateInit( )**. That is, right after most of the kernel core features are enabled. The I/O system and network are not available at this point.	**INCLUDE_USER_POST_KERNEL_APPL_INIT**
usrPreNetworkAppInit( )	Called right before the network is initialized. At this point most of the core kernel features including the I/O system are available.	**INCLUDE_USER_PRE_NETWORK_APPL_INIT**

For detailed information about the boot process, see the *VxWorks 7 BSP Developer's Guide*.

Restrictions on Use

- The stubs must be called from the **usrAppInit.c** file (in your VIP project).
- For **usrPreKernelAppInit( )**, no VxWorks APIs can be used. Use only your own code (for example, to initialize registers or light an LED).
- For **usrPostKernelAppInit( )**, most kernel APIs can be used (for creating and managing tasks, semaphores, message queues, and so on). The exceptions are those APIs that depend on the system clock—that is, those that have timeout parameter cannot be used. The I/O system and network are not available at this point.

- For **usrPreNetworkAppInit( )**, all kernel APIs can be used except for networking.

1.8 Using userAppInit( ) to Run Applications Automatically

VxWorks can be configured to start kernel applications automatically once VxWorks has booted.

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

Step 2: Add a call to the application's entry-point function to the **usrAppInit( )** function stub, which is in the **usrAppInit.c** file in your VxWorks image project (VIP) directory.

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

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

    myAppStartUp();

    }
```

(The **USER_APPL_INIT** macro is a vestigial element of the legacy build system, and is not currently used.)

Step 3: Link the kernel-base application object modules with the kernel using a DKM project.

2 Real-Time Processes: RTPs

2.1 About Real-Time Processes

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 (by default, 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).

For 64-bit VxWorks, RTP virtual memory is overlapped, and RTP applications are built as absolutely-linked executables. For 32-bit VxWorks, the default is simply RTP overlapped virtual memory, but Wind River recommends taking additional configuration steps to support RTPs as absolutely-linked executables.

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

The legacy flat RTP virtual memory configuration can also be used with 32-bit VxWorks, but only with 6.9-compatible BSPs.

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

Processes and 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 *RTPs and POSIX*, p.20).

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. RTPs are not scheduled by default—tasks are scheduled globally throughout the system.
- RTPs 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.)
- RTPs 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.
- RTP 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.
- RTPs 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 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.

RTP Ownership and Inheritance

The default owner of all RTPs and user-mode tasks is the root user, which for VxWorks has a user ID of 1 and a group ID of 1. RTPs inherit the user ID and group ID of the task that spawned them. RTP tasks inherit the user ID and group ID of their RTP, and therefore all tasks in a given RTP have the same owner. If one of the tasks changes its ownership, however, it changes the ownership of all the tasks in that RTP, as well as the RTP itself, at the same time.

The ownership of a task can be obtained with **getuid( )** and **getgid( )**, and changed (under certain conditions) using **setuid( )** and **setgid( )**. If a task changes its user ID from root to another user, it loses the ability to change it again. That is, a task that is not owned by root cannot change its user ID or group ID. If it retains the root user ID, however, it can change its group ID without restriction.

The **ps -l** shell command can be used to display the ownership (user and group IDs) of RTPs.

Support for user and group identification is included in VxWorks by default, with the **USER_IDENTIFICATION_INHERITANCE** VSB option.

For information about kernel tasks, see *6.3 Task Ownership and Inheritance*, p. 81.

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 *RTPs and Tasks*, p. 17). In the second phase, the initial process task loads the entire executable and starts the main function.

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( )** function 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. 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 *10. 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. For information about the issue of inheritance of attributes from parent processes, see *RTPs and Inheritance*, p. 18.

For information about what operations are possible on a process in each phase of its instantiation, see the API reference for **rtpLib**.

VxWorks processes can be started in the following ways:

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

Form more information in this regard, see *3.10 RTP Application Execution*, p.32.

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( )** function 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 *3.10 RTP Application Execution*, p.32.
- If the **kill( )** function 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 *13. Error Detection and Reporting*).

For information about attribute inheritance and what happens to a process' resources when it terminates, see *RTPs and Inheritance*, p.18.

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.

Each process is protected from any other process that is running on the system by the MMU. 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 function in another process, or a kernel function that is not exported as a system call.

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.

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*.

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

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( )** function, 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 functions 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 functions 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.

RTPs and Scheduling

By default, 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.

The VxWorks default scheduler does provide an optional time-sharing capability—round-robin scheduling. But this 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.

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

For more information about the default VxWorks scheduling see *6.5 Task Scheduling*, p.85.

For information about the optional RTP time partition scheduler, see *15.11 RTP Time Partition Scheduling*, p.388.

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, and *public* message queues). For more information, see *7.18 Inter-Process Communication With Public Objects*, p.147.

RTPs and Inheritance

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.

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 *3.6 RTP Environment Variables*, p.30).

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( )** function). If the kernel created the initial task, then the signal mask is zero, and all signals are unblocked.

The **getppid( )** function 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.

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.)

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 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 *7.18 Inter-Process Communication With Public Objects*, p. 147.

RTPs and Environment Variables

By default, an RTP is created without environment variables. Environment variables can be created explicitly with API calls, or by inheritance if the RTP is spawned from the kernel shell.

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.

For more information, see *3.6 RTP Environment Variables*, p.30.

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( )**
- file ownership and file permissions

For information about POSIX support, see *9. POSIX Facilities*.

VxWorks can be configured to provide POSIX PSE52 support (for individual processes, as defined by the profile). For detailed information, see *9.2 VxWorks Configuration With POSIX Facilities*, p.179.

2.2 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.

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 **wrtool** command-line tool).

Configuring VxWorks With Basic RTP Support

In order to run RTP applications on a hardware target, VxWorks must be configured with RTP support in both VSB and VIP projects.

Step 1: In your VSB project, check that **CORE_RTP** is selected (the default).

Step 2: Build the project.

Step 3: Base your VIP project on your VSB project.

Step 4: In your VIP project, select **INCLUDE_RTP** component. Doing so automatically includes other components required for RTP support.

Step 5: Build VxWorks.

Configuring VxWorks for RTPs (with **INCLUDE_RTP**) provides the basic RTP overlapped virtual memory. In addition, the components required for MMU memory protection are included. MMU support is required for RTPs.

2.3 Additional Useful VIP Components for RTPs

In addition to the component required for basic RTP support, various other components provide facilities that are useful for both development and deployed systems that support RTPs.

Components Useful for RTP Application Development

- **INCLUDE_ROMFS** for the ROMFS file system. 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).
- **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_SHL** for shared libraries.
- **INCLUDE_RTP_HOOKS** for the programmatic hook facility, which allows for registering kernel functions 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 *User-Level Clock Selection*, p.208.
- **INCLUDE_PROTECT_TASK_STACK** for stack protection. For deployed systems this component may be omitted to save on memory usage. See *6.9 Task Stack*, p.95 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.

- 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 functions.

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 *9.2 VxWorks Configuration With POSIX Facilities*, p.179). It can be used with either **BUNDLE_RTP_DEPLOY** or **BUNDLE_RTP_DEVELOP**.

3
RTP Applications

3.1 About RTP Applications

Real-time process (RTP) applications are user-mode applications similar to those used with other operating systems, such as UNIX and Linux.

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: RTPs*.

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( )**

function. 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 as absolutely linked objects, 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, depending on their use (partially-linked executables are used with shared libraries.)

During development, processes can be spawned to execute applications from the VxWorks shell or Workbench. 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. 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 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, use the **__thread** storage class instead of **tlsLib** functions, and be compiled specifically for the system in question (SMP or UP).

⚠ **CAUTION:** Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

RTP Applications and Kernel Component Requirements

RTP applications require VxWorks kernel support. For information about configuring VxWorks for RTPs, see *2.2 Configuring VxWorks for Real-time Processes*, p.20.

⚠ **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 *3.7 RTPs and Required Kernel Support*, p. 31.

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. If the file is not stored on the target system, the path must be valid from the point of view of the target itself. For information in this regard, see *11.4 Remote File System Access From VxWorks*, p. 294.

Comparing Kernel Applications with RTP Applications

The primary difference between kernel applications and RTP applications is that RTP applications execute in separate, protected memory environments, while kernel applications do not. There are also differences in the features available in each environment, as well as differences in performance, footprint, and memory use.

VxWorks kernel applications execute in kernel mode in the kernel memory space. They can be linked with the kernel or downloaded into it at runtime (as downloadable kernel modules—DKMs). Because kernel applications execute in the kernel, the system cannot protect the kernel from misbehavior on the part of applications, nor applications from one another. At runtime, VxWorks system tasks (such as **tExcTask**) and application tasks are distinguished only by their priority and the functionality of their code.

RTP applications execute as real-time processes in user mode, each in a their own protected memory space, which is separate from that of other RTP applications and from the kernel. RTP applications should be used when this protection is a key consideration for the system. RTP applications are built and stored separately from the kernel (on the host during development and on the target for deployed systems). VxWorks RTP applications are similar to applications for other operating systems such as UNIX and Linux, except that VxWorks real-time processes are designed for real-time systems (for more information in this regard, see *2. Real-Time Processes: RTPs*).

The kernel application environment and the RTP application environment have many of the same features, but also differ in what is available in each. For example, drivers can only be installed in the kernel, and PSE52 conformance can only be achieved in the RTP environment.

While there is no inherent difference in the speed of code executing in the kernel environment or the RTP environment, it can be affected by code optimization (although less so in the RTP environment) and the use of system calls (which are specific to the RTP environment). RTP applications that make notable use of system calls—which is generally the case—are slower than comparable kernel applications. In addition, VSB projects can be used to remove RTP-handling code from various VxWorks libraries (using the **Real-Time Process** option), which improves the performance of those libraries and reduces their size.

In addition, a VxWorks image configured with RTP support components will be slightly larger than one that is not. And the overall memory requirements for a system that employs RTP applications is generally larger than one that uses comparable kernel applications because RTP applications include user-side libraries (which "duplicate" those already included in the kernel).

3.2 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( )** function.

The **main( )** function can be used with the conventional **argc** and **argv** arguments, as well as the optional **envp** argument:

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

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

CAUTION: While VxWorks supports an **envp** parameter to **main( )** for RTP applications, Wind River recommends that you do not use it. The pointer can be invalidated if the RTP's environment variables array is relocated (which may occur if there is no room left in the array for new environment variables).

3.3 RTP Applications and Multitasking

If an application is multi-threaded (has multiple tasks), the developer must ensure that the **main( )** function 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 *6.1 About Tasks and Multitasking*, p.76.

3.4 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.

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

VxWorks system header files are copied to the VSB project directory, below *vsbProjectDir***/usr/h**.

CAUTION: Do not reference header files that are for kernel code in RTP applications (see *1.4 VxWorks Header Files*, p.5).

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 *User-Level Standard C Library: libc*, p.184 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 also require these definitions. Include **vxWorks.h** with the following line:

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

⚠ **CAUTION:** You must include the **vxWorks.h** header file in your RTP application before any other header files. This header file provides basic definitions and types that are used by many VxWorks facilities. It is also required by many other VxWorks header files.

The only exception to this rule is if your RTP application must conform to the POSIX PSE52 profile. In this case, do not include **vxWorks.h**. The **vxWorks.h** header file is required for non-POSIX VxWorks facilities, but it also makes applications non-conformant with the PSE52 profile.

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 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, which are located in directories named **private**. The names of the private header files have the same root as the public ones, but with a **P** added. For example, the private header file for **semLib** is **semLibP.h**.

Header Files and extern "C" Statements

Do not include system header files within **extern "C"** statement brackets. All VxWorks system header files are already configured for use with C and C++ compilers. You may invalidate declarations in the VxWorks headers if you force a C linkage on the entire header file's content. The following is incorrect:

```
#ifdef _cplusplus
extern "C" {
```

```
#endif

#include <stdio.h>
```

But the following is correct:

```
#include <stdio.h>

#ifdef _cplusplus
extern "C" {
#endif
```

In addition, do not include a VxWorks header file within an **extern "C"** statement in C++ files. The following is incorrect:

```
extern "C" {
#include <stdio.h>
}
```

But this the following is correct:

```
#include <stdio.h>
```

3.5 RTP Application APIs: System Calls and Library Functions

VxWorks provides an extensive set of APIs for developing RTP applications. As with other operating systems, these APIs include both system calls and library functions.

Some library functions 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. 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 functions and data structures. 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 functions 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.

Monitoring System Calls

The VxWorks kernel shell provides facilities for monitoring system calls. For more information, see the **syscall monitor** entry in the *VxWorks Kernel API Reference*.

VxWorks Libraries

VxWorks distributions include libraries of functions that provide APIs for RTP applications. Some of these functions execute entirely in the process in user mode. Others are wrapper functions 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( )** function performs a lot of formatting and so on, but ultimately must call **write( )** to output the string to a file descriptor.

Library functions 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 Dinkumware 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 *4.14 VxWorks Run-time C Shared Library libc.so*, p.65.

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

Custom Libraries

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

API Documentation

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

3.6 RTP Environment Variables

By default, an RTP is created without environment variables. Environment variables can be created explicitly with API calls, or by inheritance if the RTP is spawned from the kernel shell.

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( )** function 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( )** function—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 *4.6 Shared Library Location and Loading at Run-time*, p.52.)

For more information, see the **rtpSpawn( )** API reference and *3.2 RTP Application Structure*, p.26 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( )** function can be used to get the environment variables, and the **setenv( )** and **unsetenv( )** functions 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.

3.7 RTPs and 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( )** function can also be used to determine whether or not a particular system call is present in the system.

3.8 RTP Hook Functions

RTP hook functions can be used to call functions automatically when RTPs are spawned and deleted.

For information about using hook functions, which are called during the execution of **rtpSpawn( )** and **rtpDelete( )**, see the VxWorks API reference for **rtpHookLib** and *6.14 Tasking Extensions: Using Hook Functions*, p.101.

3.9 Executable File Size Reduction With the strip Facility

For production systems, it may be useful to strip executables to reduce their size. The **strip***arch* utility can be used with the **--strip-unneeded**, **--strip-debug** (or **-d**), or **--strip-all** (or **-s**) options.

Stripping Absolutely-Linked RTP Executables

All RTP executables are built as absolutely-linked 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.

The symbolic and relocation information can be stripped out of an executable using the **strip***arch* utility with the **--strip-all** (or **-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 that it may be useful to leave the symbolic and relocation information in executable files for debugging and 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.

3.10 RTP Application Execution

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 first argument of which identifies the executable. 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.

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

Ways of Starting an RTP Application

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 *3.11 Interactive Execution of RTP Applications*, p.33 and *3.12 Automatic Execution of RTP Applications*, p.35.

Ways of Terminating an RTP Application

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

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( )** function 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.

Automatic Termination Leaving Spawned Tasks Running

By default, a process is terminated when the **main( )** function 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( )** function. When **main( )** includes **taskExit( )** as its last call, the process' initial task can exit without the kernel automatically terminating the process.

3.11 Interactive Execution of RTP Applications

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.

This section describes usage from the shell command-line. For information about using Wind River Workbench facilities to run applications, see the Workbench documentation.

Starting RTP Applications from the Kernel Shell

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

NOTE: If the file is not stored on the target system, the path must be valid from the point of view of the target itself. For information in this regard, see *11.4 Remote File System Access From VxWorks*, p.294.

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

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

In this example, a process is started to run the application file **myVxApp.vxe**, which is stored on the host system **mars** in **c:/scratch**. Note that the host must be specified for non-NFS network file systems.

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.

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):

```
mars:c:/scratch/myVxApp.vxe first second third
```

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

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

NOTE: For 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.

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.

Executing Functions in an RTP Application from the Kernel Shell

The VxWorks kernel shell provides facilities for calling application functions in a real-time process. These facilities are particularly useful for debugging RTP applications. In addition, that the kernel shell provides facilities for monitoring system calls.

For more information, see the *VxWorks 7 Kernel Shell User's Guide*, and the entries for **func call, syscall monitor** and **sysCallMonitor( )** the *VxWorks 7 Kernel API Reference*.

Terminating Applications from the Kernel Shell

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 *RTP Termination*, p.16.

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

3.12 Automatic Execution of RTP Applications

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.

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( )** function

The components that support this functionality are, respectively:

- **INCLUDE_RTP_APPL_INIT_STRING**
- **INCLUDE_RTP_APPL_INIT_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( )** function 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( )** function 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.36. 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.

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.36).

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

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.38).

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, *myImageDir***/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_INIT_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_INIT_BOOTLINE** component.

(See *Specifying Applications with a Boot Loader Parameter*, p.37 and *Application Startup String Syntax*, p.36.)

The operating system must be configured with the kernel shell and the C interpreter components for use with C interpreter shell scripts.

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.

Specifying Applications with usrRtpAppInit( )

The VxWorks application startup facility can be used in conjunction with the **usrRtpAppInit( )** initialization function to start applications automatically when VxWorks boots.

In order to use this method, VxWorks must be configured with the **INCLUDE_RTP_APPL_USER** component, which includes the file in your project automatically.

For each application you wish to start, add an **rtpSpawn( )** call and associated code to the **usrRtpAppInit( )** function stub.

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";
```

3.13 Applications and Symbol Registration

Symbol registration is the process of storing symbols in a kernel-based symbol table that is associated with a given RTP.

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 /usr/xeno/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
```

For information about stripping symbols from RTP executables, see *3.9 Executable File Size Reduction With the strip Facility*, p.32.

3.14 Caution About Using Show Functions With RTP Applications

Show functions and shell commands cannot be used to display information about RTP-side objects. A number of information retrieval functions such as **semInfoGet( )** and **msgQInfoGet( )** are, however, available in the RTP environment. They can be used programmatically to retrieve RTP-side information about a number of RTP object categories.

Show functions cannot display information about an RTP object if they are passed an RTP object identifier because they execute in the kernel environment. If, for example, debugging code is used in an RTP application to display the identifier of a given object such as a semaphore, then using the RTP object identifier with a show function such as **semShow( )** from the shell causes an error (with an "**Invalid** *objectName* **id**" error message).

Note that some RTP-side object categories have both an RTP-side object and a corresponding kernel-side object. In these cases, only the identifier of the RTP-side object can be used from within an RTP, and only the kernel-side identifier can be used from within the kernel. This means that show functions and shell commands such as **rtpShow( )**, **i( )**, and **w( )** display kernel object identifiers that relate to RTP objects, which may be a cause for confusion. For example, **i( )** displays kernel identifiers for RTP-side tasks, and **w( )** displays kernel identifiers for RTP-side semaphores and message queues.

Other RTP-side object categories—memory partition identifiers for example—do not have a kernel-side counterpart. Furthermore, some RTP-side object categories may internally use other objects that do have a kernel-side counterpart. For example, POSIX threads are based on VxWorks native tasks and thus use a task object internally.

4
RTP Libraries and Plug-Ins

4.1 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 *4.11 RTP Application Development for use With Shared Libraries*, p.60 and *4.13 RTP Application Development for use With Plug-Ins*, p.61.

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 functions 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 *4.7 Lazy Binding and Shared Libraries*, p.55.)

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 functions 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.

4.2 VxWorks Configuration 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.

VIP Components

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 functions 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.22).

For general information about configuring VxWorks for real-time processes, see *4.2 VxWorks Configuration for Shared Libraries and Plug-ins*, p.46.

4.3 Common Development Issues for Initialization and Termination

Development of static libraries, shared libraries, and plug-ins all share some common issues of initialization and termination, which must be considered in their design and development.

Library and Plug-in Initialization

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

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

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

The function 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 function 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 function (or functions), as well as the order in which they should be called. The macro takes two arguments, the name of the function and a rank number. The function itself makes up the body of the macro. The syntax is as follows:

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

The following example is of a function 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 *13.7 Other Error Handling Options for RTPs*, p.359.)

The rank number is used by the compiler to order the initialization functions. (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 functions 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 functions 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).

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 *Library and Plug-in Initialization*, p.46).

By default, static constructors are called last, after the library's (or plug-in's) initialization function. 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 function if the objects and methods defined by the library (or plug-in) are properly designed (using deferred initialization).

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 function.
- 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 *13.7 Other Error Handling Options for RTPs*, p.359.

Shared Library and Plug-in Removal

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 Functions

There is no library (or plug-in) *termination* function facility comparable to that for initialization functions (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( )** function 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 function. Cleanup functions registered with **atexit( )** are called when **exit( )** is called. Note that if a process' task directly calls the POSIX **_exit( )** function, none of the cleanup functions 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 functions are executed in reverse order of their registration when a process is being terminated.

⚠ **CAUTION:** Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

4.4 Static Library Development

Static libraries (archives) are made up of functions 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.

4.5 Shared Library Development

Shared libraries are made up of functions 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 run-time dynamic linker to locate and load the shared library for the application automatically.

Once loaded into memory by the dynamic linker, shared libraries are held in sections of memory 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.

Use a shared library project with **wrtool** or Workbench to create a shared library.

⚠ **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 *4.11 RTP Application Development for use With Shared Libraries*, p. 60.

⚠ **CAUTION:** Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

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

Dynamic Linker

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.

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.

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.

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.52)

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 *Creating Shared Object Names for Shared Libraries*, p.51 and *4.11 RTP Application Development for use With Shared Libraries*, p.60. For examples of displaying ELF records (including **SONAME** and **NEEDED**), see *Using readelf to Examine Dynamic ELF Files*, p.58.

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 *About Shared Library Names and ELF Records*, p.50. For information about identifying the runtime location of shared libraries, see *4.6 Shared Library Location and Loading at Run-time*, p.52.

Note that a plug-in does not require a shared object name. For information about plug-ins, see *4.12 Plug-In Development*, p.61.

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 *Using Different Versions of Shared Libraries*, p.52.

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.58.

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 *Creating Shared Object Names for Shared Libraries*, p.51.

4.6 Shared Library Location and Loading 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.53.
2. The configuration file **ld.so.conf**. For more information, see *Using the ld.so.conf Configuration File*, p.54.
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.54.
4. The same directory as the one in which the application file is located. For more information, see *Using the Application Directory*, p.54.

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.55.

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.52). 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 *RTPs and Environment Variables*, p.19.

NOTE: If the file is not stored on the target system, the path must be valid from the point of view of the target itself. For information in this regard, see *11.4 Remote File System Access From VxWorks*, p.294.

NOTE: For Windows hosts must use forward slashes (or double backslashes) as path delimiters. This is the case even when the executable is stored on the host system.

Setting LD_LIBRARY_PATH From the Shell

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
```

The command for shared libraries stored in an NFS directory would look very similar, with the path starting with the mount point on VxWorks. For information about NFS, see *11.4 Remote File System Access From VxWorks*, p.294 and the *VxWorks 7 File Systems Programmer's Guide: Network File System*).

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

```
rtp exec -e
"LD_LIBRARY_PATH=mars:c:/vsbProjDir/usr/root/diab/bin;
mars:c:/wrwb_demo/RtpAppShrdLib/lib/SIMPENTIUMdiab"
mars:c:/wrwb_demo/RtpAppShrdLib/app/SIMPENTIUM/bin/myVxApp.vxe one two three
```

Note that in this example the path includes the host name (in this case **mars**), which is required for non-NFS network file systems. For more information in this regard, see *11.4 Remote File System Access From VxWorks*, p.294.

Setting LD_LIBRARY_PATH From Application Code

The following code fragment illustrates setting **LD_LIBRARY_PATH** from application code, which would be necessarily be done before any RTP application that required the specified library was spawned:

```
#include <envLib.h>
...
putenv("LD_LIBRARY_PATH=/nfsdir/foo/libs");
```

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.52).

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 *RTPs and Environment Variables*, p.19.

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.52).

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 Diab Compiler and the Wind River GNU compiler. For the latter, an equal sign (=) must be used between **-rpath** and the path name.

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.52), 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.55).

For information about environment variables, see *RTPs and Environment Variables*, p.19.

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.

4.7 Lazy Binding and 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 functions and data to which a dynamic shared object refers when it loads the object. The dynamic linker can save some work in computing function addresses by delaying the computation until a function is called for the first time. If a function 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 functions 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.62.

4.8 Runtime Information About Shared Libraries

RTP applications can be started from the shell, and run in the background, so that shell commands can be used to print information about shared library use.

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
```

The **shl info** command provides the full path.

4.9 Problems With Shared Library Use and Correction

Problems with the interaction between shared libraries and the applications that use them can occur for a variety of reasons. The **readelf** tool can be useful in correcting them.

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 (*4.6 Shared Library Location and Loading at Run-time*, p.52). 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 *3.13 Applications and Symbol Registration*, p.40.)

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 **-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 *4.6 Shared Library Location and Loading at Run-time*, p.52 and *Using the ELF RPATH Record*, p.54.

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:\somePlace>readelfpentium -d 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 *Creating Shared Object Names for Shared Libraries*, p.51, *4.6 Shared Library Location and Loading at Run-time*, p.52, and *Using Different Versions of Shared Libraries*, p.52.

4.10 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 HostFs 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.

4.11 RTP Application Development for use With 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.

If an application has symbols that will be referenced by a shared library, then the application must be linked at build time in such a way that the dynamic linker registers all the symbols with the shared library's dynamic symbol table. The shared library then has access to these symbols when it is subsequently loaded. If it is not linked in this manner, then the shared library will not have access to all the symbols that it requires, and a runtime error will be generated.

Calls made between C and C++ code require special consideration, for information in this regard, see *5.5 Calls Between C and C++ Code*, p.71.

For information about the various options for locating a shared library at run-time—including using a compiler option—see *4.6 Shared Library Location and Loading at Run-time*, p.52.

Compiler Option for Dynamic Executable

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

- **-Xdynamic** for the Wind River Diab 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 *4.6 Shared Library Location and Loading at Run-time*, p.52).

4.12 Plug-In Development

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.

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 plug-in and application at run time. For more information, see *4.13 RTP Application Development for use With Plug-Ins*, p.61.

CAUTION: Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

4.13 RTP Application Development for use With Plug-Ins

The key differences between developing an RTP application that uses a plug-in and an RTP application that uses a shared library involve coding the application to load the plug-in, and how the application and plug-in are linked.

The differences 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 *4.6 Shared Library Location and Loading at Run-time*, p.52.

Code Requirements For RTP Applications That Use Plug-Ins

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.63 and *Example RTP Application That Uses a Plug-In*, p.63.

Note that calls made between C and C++ code require special consideration, for information in this regard, see *5.5 Calls Between C and C++ Code*, p.71.

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 RTP Application That Uses a Plug-In*, p.63).

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 *4.6 Shared Library Location and Loading at Run-time*, p.52.

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 *4.7 Lazy Binding and Shared Libraries*, p.55.

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 RTP Application That Uses 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 function 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 Diab 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( )** function 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( )** function looks up the protocol-specific **send_packet( )** function inside the plug-in and calls it with the address and data parameters. To unload a protocol module, the application calls **detach( )**.

Functions for Managing Plug-Ins

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

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

Function	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*.

Build Requirements For RTP Applications That Use Plug-Ins

Compile any application that uses a plug-in as a dynamic executable—that is, with the Wind River Diab Compiler **-Xdynamic** option or the GNU **-non-static** option. This 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).

If the application has symbols that are referenced by a plug-in, use the following linker option to force the dynamic linker to register the application's symbols when it is loaded:

- **-Wl,-E** or **-Wl,-Xexport-dynamic** with the Wind River Diab toolchain. Note that **-E** or -**Xexport-dynamic** can be used directly with the linker, but they require -Wl to be passed through to the linker by the compiler.
- **-E** or **-export-dynamic** with the GNU toolchain.

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).

4.14 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**
- **memEdrLib**
- networking libraries

All dynamic executables require **libc.so.1** at run time. When generating a dynamic executable, the GNU and Wind River Diab toolchains automatically use the corresponding build-time shared object, **libc.so**.

To build the **libc.so** and **libc.so.1** files, select the **LANG_LIB_LIBC_LIBC_USER** option in your VSB project.

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 *4.6 Shared Library Location and Loading at Run-time*, p.52.

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 About C++ Support for VxWorks

C++ support is provided with the Wind River Diab and the GNU toolchains. By default C++03 support is provided for both kernel and RTP applications. C++11 support is also provided as an alternative for RTP applications and shared libraries.

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. No special flags are required to enable the compilers to find these headers.

SDA and C++ Code Caveat

Small data area (SDA) relocation, which is supported by PowerPC, does not work with C++ code. Do not use the Wind River Diab compiler **-Xsmall-const=8 -Xsmall-data=8** options or the GNU **-G8 -msdata=eabi** options.

5.2 C++ Compiler Differences

The Wind River Diab C++ Compiler uses the Edison Design Group (EDG) C++ front end. It fully complies with the ANSI C++ Standard. The GNU compiler supports most of the language features described in the ANSI C++ Standard.

For complete documentation on the Diab compiler and associated tools, see the Wind River Diab Compiler documentation set. For complete documentation on the GNU compiler and on the associated tools, see the *GNU ToolKit User's Guide*.

WARNING: Wind River Diab Compiler C++ and GNU C++ binary files are not compatible. If VxWorks includes C++ code that will be used by a kernel application (linked or downloaded), then both must be built with the same toolchain. There is, however, no requirement that the same compiler be used if they only contain C code. And there is no requirement that VxWorks and RTP applications use the same compiler, whether or not they contain C++ code.

Standard Template Library (STL)

The GNU STL port for VxWorks is thread safe at the class level.

The Diab STL is thread safe.

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 Diab Compiler and the GNU compiler generate a definition in every file that needs it and put each such definition in its own section. The Diab compiler uses *COMDAT* sections for this purpose, while the GNU compiler uses *linkonce* sections. In each case the linker removes duplicate sections, with the effect that the final executable contains exactly one copy of each needed entity.

NOTE: For kernel code, only the Wind River Diab Compiler linker can be used to process files containing COMDAT sections. Similarly only the GNU linker can be used on files containing linkonce sections. Furthermore the VxWorks target and host loaders are not able to process COMDAT and linkonce sections. A fully linked VxWorks image will not contain any COMDAT or linkonce sections. However intermediate object files compiled from C++ code may contain such sections. To build a downloadable C++ module, or a file that can be processed by any linker, you must perform an intermediate link step using the **-r5** option (Diab) or specifying the **link.OUT** linker script (GCC). (Note that while the **-r5** and **-r4** options—the latter referred to elsewhere in this chapter—both collapse COMDAT files, their overall purpose is different, and their use is mutually exclusive in a single linker command.)

Wind River recommends 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 Diab Compiler and Multiple Instantiations of Templates

The Wind River Diab Compiler C++ options controlling multiple instantiation of templates are:

-Xcomdat
: This option is the default. When templates are instantiated implicitly, the generated **code** or **data** section are marked as **comdat**. The linker then collapses identical instances marked as such, into a single instance in memory.

⚠ **CAUTION:** If code is going to be used as downloadable kernel modules, the **-r4** option must be used to collapse any COMDAT sections contained in the input files.

-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 and Multiple Instantiations of Templates

The GNU C++ compiler options controlling multiple instantiation of templates are:

-fimplicit-templates
: This option is the default. Template instantiations and out-of-line copies of **inline** functions are put into special *linkonce* sections. Duplicate sections are merged by the linker, so that each instantiated template appears only once in the output file.

⚠ **CAUTION:** The VxWorks dynamic loader does not support *linkonce* sections directly for kernel modules. Instead, the linkonce sections must be merged and collapsed into standard **text** and **data** sections before loading.

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

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 Diab 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.

5.3 VxWorks Configuration for C++

By default, VxWorks includes only minimal C++ support. You can add C++ functionality by including additional components.

Additional VIP Components for C++

INCLUDE_CTORS_DTORS
(included in default kernels)
Ensures that compiler-generated initialization functions, including initializers for C++ static objects, are called at kernel start up time.

INCLUDE_CPLUS
Includes basic support for C++ applications. Typically this is used in conjunction with **INCLUDE_CPLUS_LANG**.

INCLUDE_CPLUS_LANG
Includes support for C++ language features such as **new**, **delete**, and exception handling.

INCLUDE_CPLUS_IOSTREAMS
Includes all library functionality.

INCLUDE_CPLUS_DEMANGLER
Includes the C++ demangler, which is useful if you are using the kernel shell loader because it provides for returning demangled symbol names to kernel shell symbol table queries. This component is added by default if both the **INCLUDE_CPLUS** and **INCLUDE_SYM_TBL** components are included in VxWorks.

See also *5.6 Using C++11 With RTPs and Shared Libraries*, p.71.

5.4 Required Task Spawn Option for Tasks That Use C++

Any VxWorks task that uses C++ must be spawned with the **VX_FP_TASK** option.

By default, tasks spawned from Workbench automatically have **VX_FP_TASK** enabled.

CAUTION: 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. C++ exception handlers use floating point registers. The **VX_FP_TASK** option ensures that floating point registers are saved and restored properly during context switching.

5.5 Calls Between C and C++ Code

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"**.

For example:

```
#ifdef __cplusplus
extern "C" void myEntryPoint ();
#else
void myEntryPoint ();
#endif
```

Use this syntax to make C symbols accessible to C++ code as well. VxWorks C symbols are automatically available to C++ because the VxWorks header files use this mechanism for declarations.

Using the **#ifdef** construct allows a single header file to be used for both C and C++.

Kernel Code

For kernel code, the build system produces an error when **extern "C"** is not used appropriately for external linkage between C++ and C code.

RTP Application Code

For RTP application code, if you do not use the **extern "C"** linkage specifier appropriately, the type of error that results depends on the context in which it occurs (build-time static linkage, run-time loading and linkage of a plug-in with an RTP application, and so on).

5.6 Using C++11 With RTPs and Shared Libraries

By default C++03 libraries are used for RTP applications and shared libraries. C++11 libraries are also provided, for use with the GNU compiler. When creating your RTP application or shared library, select the build specification that corresponds to your desired C++ library type.

For detailed information about C++11, see the *Dinkum C++11 Libraries Reference*.

To use C++11 with RTP applications and shared libraries, you must create your projects and configure VxWorks as follows:

Step 1: Create your VSB project.

Step 2: Configure your VSB project as follows:

1. For the primary tool option, select GNU, with **PRI_TOOL_***processor_arch_version***_gnu**.
2. For the applications tool option, select the same as the primary tool, with **APP_PRI_***archName*.
3. For the RTP C++ library option, select **LANG_LIB_CPLUS_CPLUS_USER_2011**.

Standard libraries (**STANDARD_STL**) are then selected automatically.

NOTE: C++11 support is also available when the secondary compiler is set to GNU, regardless of your primary compiler selection. However, C++11 support is not available in VxWorks 6.9 compatibility mode.

Step 3: Build the source code.

Step 4: Create your VIP project, basing it on your VSB project.

Step 5: Add **INCLUDE_CPLUS** and any other C++ components required to support your application.

For more information, see *5.3 VxWorks Configuration for C++*, p.70.

Step 6: Build VxWorks.

Step 7: Create your RTP or shared library project, basing it on your VSB project.

In doing so, ensure that the active build specification for the project is set to C++11 (*vsbName_archName*__gnu_C++2011__).

Step 8: Build your RTP application using GNU.

5.7 C++ 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.

Code Examples

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;
        }
```

5.8 Using C++ Exception Handling

C++ exception handling is provided by the **INCLUDE_CPLUS_LANG** component, but it can be disabled for user code.

To disable exception handling for user code, do the following:

Step 1: Define the environment variable **CXX_STL_VARIANT=abridged** before creating the VxWorks Image Project (VIP).

This ensures that the correct (exception-disabled) libraries are linked against.

Step 2: Add **-mc++-abr** for GNU or **-Xc++-abr** for Diab to the compiler options.

6
Multitasking

6.1 About Tasks and Multitasking

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.

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.

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. On an SMP system, multitasking involves the actual concurrence of many threads of execution.

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 properties like its user and group IDs, a thread of execution (the task's program counter), CPU registers, stacks for dynamic variables and function calls, and so on.

With few exceptions, the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few functions. Also note that some programming practices—such as implicit synchronization techniques relying on task priority instead of explicit locking—are not appropriate for an SMP system.

For information about SMP programming, see *18. VxWorks SMP*.

6.2 VxWorks System Tasks

Depending on its configuration, VxWorks starts a variety of system tasks at boot time, some of which are always running.

Basic VxWorks Tasks

The set of tasks associated with a basic configuration of VxWorks, and a few of the tasks associated with commonly used optional components, are described below.

⚠ **CAUTION:** Do not suspend, delete, or change the priority of any of these tasks. Doing so will result in unpredictable system behavior.

Root Task

Task Name	**tRootTask**
Priority	0

Triggering Event	System startup.
Component	N/A
Description	The root task is the first task executed by the kernel. The entry point of the root task is **usrRoot( )**, which initializes most VxWorks facilities. It spawns such tasks as the logging task, the exception task, the network task, and the **tRlogind** daemon. Normally, the root task terminates after all initialization has completed.
References	*VxWorks 7 BSP and Driver Guide.*

Logging Task

Task Name	**tLogTask**
Priority	0
Triggering Event	A **logMsg( )** call from either an ISR context or task context.
Component	**INCLUDE_LOGGING**
Description	The log task is used by VxWorks modules to log system messages and messages from an ISR without having to perform I/O in the current task context.
References	*11.8 Asynchronous Input/Output*, p.313 and the **logLib** API reference.

Exception Task

Task Name	**tExcTask**
Priority	0
Triggering Event	An **excJobAdd( )** call from ISR context.
Component	**INCLUDE_EXC_TASK**
Description	The exception task executes jobs—that is, function calls—on the behalf of ISRs, as they cannot be performed at interrupt level. It must have the highest priority in the system.
References	The **excLib** API reference entry.

Job Task

Task Name	**tJobTask**
Priority	0 while waiting for a request.
Triggering Event	N/A. All jobs are queued by VxWorks system facilities.
Component	**INCLUDE_JOB_TASK**

Description	The job task executes jobs—that is, function calls—on the behalf of tasks. It runs at priority 0 while waiting for a request, and dynamically adjusts its priority to match that of the task that requests job execution. One of its primary purposes is to handle *suicidal* task deletion (that is, self-deletion of a task).
References	*6.13 Task Deletion and Deletion Safety*, p.100.

SMP ISR Task

Task Name	**tIsr***N*
Priority	0
Triggering Event	A device interrupt that calls **isrDeferJobAdd( )**.
Component	**INCLUDE_ISR_DEFER**
Description	The **tIsr***N* task (or tasks) execute function calls on behalf of device drivers when those drivers execute **isrDeferJobAdd( )**. The *N* in the task name refers to the index of the CPU on which the deferral task is running. Separate **tIsr***N* tasks are created as needed, each with CPU affinity to CPU *N*. SMP-aware device drivers defer interrupt-level processing to the **tIsr***N* task that is executing on the local CPU, to avoid the overhead of cross-processor communication to schedule the deferral task.
References	The **isrDeferLib** API reference entry.

Network Task

Task Name	**tNet0**
Priority	50 (default).
Triggering Event	Packet arrival, transmit completion, network protocol timer expiration, socket application request, and so on.
Component	**INCLUDE_NET_DAEMON**
Description	The network daemon **tNet0** performs network driver and network protocol processing for the VxWorks network stack.
References	*Wind River Network Stack Programmer's Guide*.

Tasks for Optional Components

The following tasks are examples of some of the additional tasks found in common configurations of VxWorks.

Kernel Shell Task

Task Name	**tShell***N*
Priority	1 (configurable)
Triggering Event	System boot (default).
Component	**INCLUDE_SHELL**
Description	The kernel shell is spawned as a task. Any function or task that is invoked from the kernel shell, rather than spawned, runs in the **tShell***num* context. The task name for a shell on the console is **tShell0**. The kernel shell is re-entrant, and more than one shell task can run at a time (hence the number suffix). In addition, if a user logs in remotely (using **rlogin** or **telnet**) to a VxWorks target, the name reflects that fact as well. For example, **tShellRem1**. The **tShell** basename is configurable, see the *VxWorks Kernel Shell User's Guide*.
References	*VxWorks 7 Kernel Shell User's Guide*.

Kernel Shell Login Task

Task Name	**tRlogind**
Priority	55
Triggering Event	New rlogin connection.
Component	**INCLUDE_RLOGIN**
Description	The kernel shell login daemon allows remote users to log in to VxWorks. It accepts a remote login request from another VxWorks or host system and spawns **tRlogInTask_***hexNum* and **tRlogOutTask_***hexNum* (where *hexNum* is a hexadecimal number identifying the connection). These tasks exist as long as the remote user is logged on. In addition, unless the shell is configured in VxWorks 5.5 compatibility mode, the server spawns a remote shell task **tShellRem***decNum* (where *decNum* is a decimal number specific to the particular remote shell session).
References	The **rlogLib** API reference entry, the *Wind River Network Stack Programmer's Guide*, and the *VxWorks 7 Kernel Shell User's Guide*.

Kernel Shell Telnet Task

Task Name	**ipcom_telnetd**
Priority	50
Triggering Event	New telnet connection.

Component	**INCLUDE_IPTELNETS**
Description	This daemon allows remote users to log in to a VxWorks kernel shell using telnet. It accepts remote login requests from another network host. **ipcom_telnetd** spawns other tasks to handle each incoming connection, including **ipcom_telnetspawn**, **tStdioProxy***hexNum*, and **tLogin***hexNum* (where *hexNum* is a hexadecimal number that identifies the particular connection). Unless the shell is configured in VxWorks 5.5 compatibility mode, a remote shell task **tShellRem***decNum* is also spawned (where *decNum* is a decimal number specific to the particular remote shell session).
References	*Wind River Network Stack Programmer's Guide*.

RPC Task

Task Name	**tPortmapd**
Priority	54
Triggering Event	Query by a client to look up an RPC service.
Component	**INCLUDE_RPC**
Description	This daemon is RPC server that acts as a central registrar for RPC services running on the same machine. RPC clients query the daemon to find out how to contact the various servers.
References	The **rpcLib** API reference entry.

TCF Task

Task Name	**tTcf**
Priority	49
Triggering Event	A blocking job requested by the **tTcfEvent** task.
Component	**INCLUDE_DEBUG_AGENT**
Description	**tTcf** tasks are worker tasks used to execute blocking operations. As the **tTcfEvent** task cannot be blocked, all blocking operations are deferred to these tasks. As soon as the job is finished, the result is sent to the **tTcfEvent** task to process the data.

TCF Events Task

Task Name	**tTcfEvents**
Priority	49

Triggering Event	A TCF client request or a target event.
Component	**INCLUDE_DEBUG_AGENT**
Description	The **tTcfEvents** task handles all target events (such as context creation and destruction, and breakpoints). It also handles all requests that comes from TCF clients (the debugger, debug shell, and so on). Blocking operations requested by an event (such as reading a file descriptor) are deferred to a pool of worker tasks that send the result back to the **tTcfEvents** task in the form of an event.

Debug Task

Task Name	**tVxdbgTask**
Priority	25
Triggering Event	A VxWorks context event that must be reported to the debug tools.
Component	**INCLUDE_VXDBG**
Description	This task provides the interface to the event support provided by the VxWorks debug library (VxDBG event library). This support consists of a task-level event deferral facility. By definition, an event is a callback function for a VxWorks context event (such as a task or RTP creation event, task or RTP task stop event, or task exception event) that must be reported to debug tools (kernel shell, Workbench debugger, and so on). Deferred callback executions are asynchronous.

6.3 Task Ownership and Inheritance

The root user is the default owner for all tasks and RTPs. Inheritance of user and group IDs is different for kernel and RTP tasks. The IDs can be changed.

Support for user and group identification is included in VxWorks by default, with the **USER_IDENTIFICATION_INHERITANCE** VSB option.

The default owner of all tasks (both kernel and RTP) is the root user, which for VxWorks has a user ID of 1 and a group ID of 1. The root user is also the default owner of RTPs. Kernel tasks inherit the user ID and group ID from their parent task—the task that spawned them. RTPs inherit the user ID and group ID of the task that spawned them. RTP tasks inherit the user ID and group ID of their RTP, and therefore all tasks in a given RTP have the same owner. If one of the tasks changes its ownership, however, it changes the ownership of all the tasks in that RTP, as well as the RTP itself, at the same time.

The ownership of a task can be obtained with **getuid( )** and **getgid( )**, and changed (under certain conditions) using **setuid( )** and **setgid( )**. If a task changes its user ID from root to another user, it loses the ability to change it again. That is, a task that is not owned by root cannot change its user ID or group ID. If it retains the root user ID, however, it can change its group ID without restriction.

The **taskShow( )** function can be used to display tasks user IDs and group IDs, by using 0 and 5 as the parameters. The **ps -l** shell command can be used to display the ownership (user and group IDs) of RTPs.

⚠ **CAUTION:** Note that there is no privilege verification system provided for tasks or RTPs as such. If you develop your own using the APIs provided, be careful that it does not interfere with VxWorks features that manage user permissions (see *16User Authentication and Management*, p.399).

6.4 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.87.

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).

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

State Symbol	Description
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.
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 *13. Error Detection and Reporting*).

Example of Task States in Shell Command Output

This example shows output from the **i( )** shell command, displaying task state information.

```
-> i
NAME          ENTRY        TID      PRI   STATUS      PC       SP      ERRNO  DELAY
----------  ------------ -------- --- ---------- -------- -------- ------- -----
tJobTask    601f3bb0     603ec010   0 PEND       602a22bb 60be1f30       0     0
tExcTask    601f35e0     603a4e40   0 PEND       602a22bb 603a4d40       0     0
tLogTask    logTask      603ec3e0   0 PEND       6029fdfb 60befec0       0     0
tShell0     shellTask    6058d2a0   1 READY      602aaf60 60d47c50       0     0
ipcom_tick> 600d4110     60d4dc30  20 PEND       602a22bb 60c75f10       0     0
tVxdbgTask  601900b0     60425848  25 PEND       602a22bb 60ce7f40       0     0
```

```
tTcfEvents  602511e0     60388560  49 PEND+T      602a22bb 603b2210       0   537
tTcf        601a98e0     60590fc0  49 PEND        602a22bb 60d5b8f0       0     0
tTcf        601a98e0     60591c80  49 PEND        602a22bb 60d6f8f0       0     0
tTcf        601a98e0     605951e0  49 PEND        602a22bb 60d83920       0     0
tAioIoTask> aioIoTask    60404338  50 PEND        602a2ab6 60c15f20       0     0
tAioIoTask> aioIoTask    60408010  50 PEND        602a2ab6 60c27f20       0     0
tNet0       ipcomNetTask 60489b30  50 PEND        602a22bb 60c35f20       0     0
ipcom_sysl> 600d2dd0     60c6dac0  50 PEND        602a2ab6 60c89e50       0     0
tNetConf    6010d980     603fdac8  50 PEND        602a22bb 60cc5d60       0     0
ipcrypto_r> ipcrypto_rn> 60419618  50 DELAY       602a9b97 60cd9f80       0    37
tAnalysisA> cafe_event_> 60374600  50 PEND        602a22bb 6039a2b0       0     0
tAioWait    aioWaitTask  60404010  51 PEND        602a22bb 60c03e90       0     0
ipcom_egd   600d1eb0     604201d0 255 DELAY       602a9b97 60c97ea0       0    37
value = 0 = 0x0
->
```

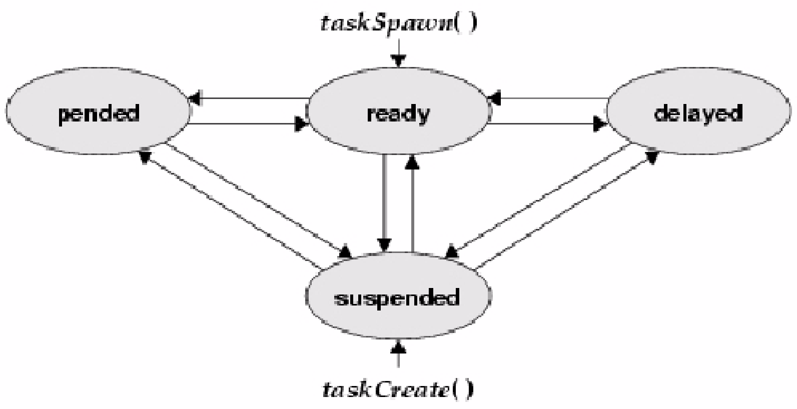
Illustration of Basic Task State Transitions

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.82, nor does it show the **STOP** state used by debugging facilities.

The functions 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( )

6.5 Task Scheduling

Multitasking requires a task scheduler to allocate the CPU to ready tasks. VxWorks provides several scheduler options.

The options are as follows:

- The traditional VxWorks scheduler, which provides priority-based, preemptive scheduling, as well as a round-robin extension. See *VxWorks Traditional Scheduler*, p.86.
- The VxWorks POSIX threads scheduler, which is designed (and required) for running pthreads in processes (RTPs). See *9.18 POSIX and VxWorks Scheduling*, p.208.)
- The RTP time partition scheduler, which allows for scheduling RTPs themselves for specified time frames. See *15.11 RTP Time Partition Scheduling*, p.388.
- A custom scheduler framework, which allows you to develop your own scheduler. See *20. Custom Scheduler*.

Task Priorities

Task scheduling relies on a task's priority, which is assigned when it is created. The VxWorks kernel provides 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.

While a task is assigned its priority at creation, you can change it programmatically thereafter. For information about priority assignment, see *6.6 Task Creation and Activation*, p.90 and *6.12 Task Scheduling Control*, p.99).

Kernel Application Task Priorities

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

NOTE: Network application tasks may need to run at priorities lower than 100. For information in this regard, see the *VxWorks Network Stack Programmer's Guide*.

Driver Task Priorities

In contrast to kernel application tasks, which should be in the task priority range from 100 to 255, driver *support* tasks (which are associated with an ISR) can be in the range of 51-99.

These tasks are crucial; for example, if a support task fails while copying data from a chip, the device loses that data. Examples of driver support tasks include **tNet0** (the VxWorks network daemon task), an HDLC task, and so on.

The system **tNet0** has a priority of 50, so user tasks should not be assigned priorities below that task; if they are, the network connection could die and prevent debugging capabilities with the host tools.

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 RTP time partition scheduler, the POSIX thread scheduler, and custom schedulers, see *15.11 RTP Time Partition Scheduling*, p.388, *9.18 POSIX and VxWorks Scheduling*, p.208, and *20. Custom Scheduler*, respectively.

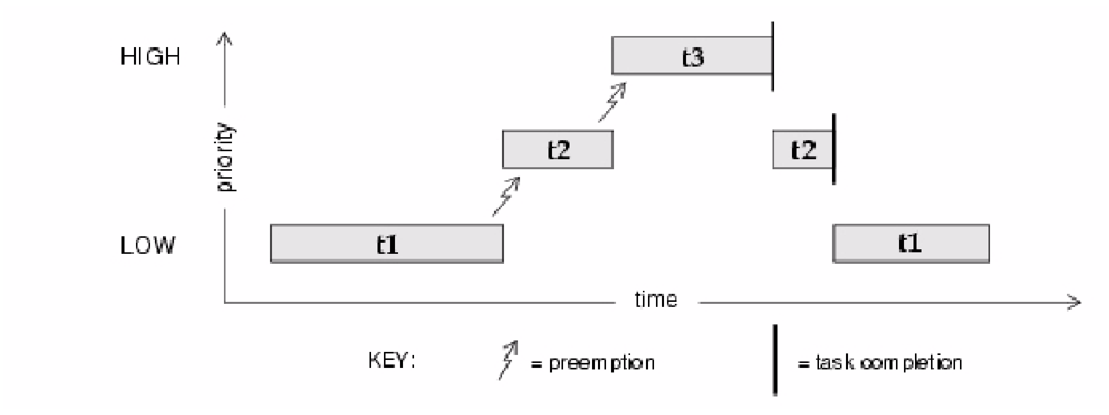
Priority-Based Preemptive Scheduling

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.88).

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 *6.4 Task States and Transitions*, p.82).
- 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.124.)

Task Rotation: Shifting a Kernel Task to End of Priority List

The **taskRotate( )**function can be used shift a kernel task from the front to the end of its priority list in the ready queue. For example, the following call shifts the task that is at the front of the list for priority level 100 to the end:

```
taskRotate(100);
```

To shift the task that is currently running to the end of its priority list, use **TASK_PRIORITY_SELF** as the parameter to **taskRotate( )** instead of the priority level.

The **taskRotate( )** function can be used as an alternative to round-robin scheduling. It allows a program to control sharing of the CPU amongst tasks of the same priority that are ready to run, rather than having the system do so at predetermined equal intervals. For information about round-robin scheduling, see *Round-Robin Scheduling*, p.88.

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.87).

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.

Round-Robin Scheduling and Kernel Applications

For kernel applications, round-robin scheduling is not necessary for systems in which all tasks run at different priority levels. It is designed for systems in which multiple tasks run at the same level.

Note that the **taskRotate( )**function can be used as an alternative to round-robin scheduling. It is useful for situations in which you want to share the CPU amongst tasks of the same priority that are ready to run, but to do so as a program requires, rather than at predetermined equal intervals. For more information, see *Task Rotation: Shifting a Kernel Task to End of Priority List*, p.87.

Round-Robin Scheduling and RTP Applications

For RTP applications, 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.

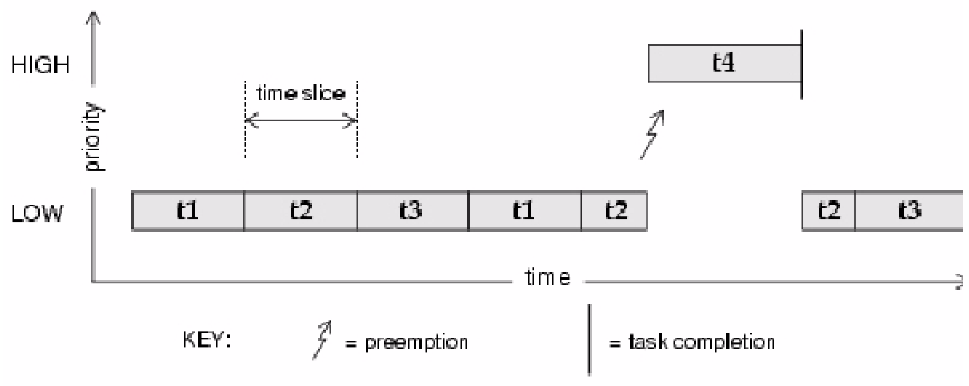
Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.

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**



6.6 Task Creation and Activation

The functions the **taskLib** library provide the means for task creation and control, as well as for retrieving information about tasks.

The functions listed in Table 6-2 are used to create tasks.

The arguments to **taskSpawn( )** are the new task's name (an ASCII string), the task's priority, options, the stack size, the main function address, and 10 arguments to be passed to the main function 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 *6.12 Task Scheduling Control*, p.99.

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

Table 6-2 **Task Creation Functions**

Function	Description
taskSpawn( )	Spawns (creates and activates) a new task.
taskCreate( )	Creates, but not activates a new task.
taskInit( )	Initializes a new task (kernel only).
taskInitExcStk( )	Initializes a task with stacks at specified addresses (kernel only).
taskOpen( )	Open a task (or optionally create one, if it does not exist).
taskActivate( )	Activates an initialized task.

The **taskOpen( )** function 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 a public object, with visibility across all processes and the kernel (see *Public Naming of Tasks for Inter-Process Communication*, p.93). The **taskOpen( )** function is the most general purpose task-creation function.

The **taskSpawn( )** function embodies the lower-level steps of allocation, initialization, and activation. The initialization and activation functions are provided by the functions **taskCreate( )** and **taskActivate( )**—however, Wind River recommends that you use these functions only when you need greater control over allocation or activation.

The difference between the kernel functions **taskInit( )** and **taskInitExcStk( )** is that the **taskInit( )** function allows the specification of the execution stack address, while **taskInitExcStk( )** allows the specification of both the execution and exception stacks.

Static instantiation of Kernel Tasks

The task creation functions listed in Table 6-2 perform a dynamic, two-step operation, in which memory is allocated for the task object at runtime, and then the object is initialized. Kernel tasks (and other VxWorks objects) can also be statically instantiated—which means that their memory is allocated for the object at compile time—and the object is then initialized at runtime.

The **VX_TASK** macro declares a task object at compile time. The macro takes two arguments: the task name and its stack size. Unlike **taskSpawn( )** use, with which the name may be a **NULL** pointer, the name is mandatory with the **VX_TASK** macro. The stack size must evaluate to a non-zero integer value and must be a compile-time constant.

The **VX_TASK_INSTANTIATE** macro can be used with **VX_TASK** to initialize and schedule a task statically instead of using its dynamic equivalent, **taskSpawn( )**. Alternatively, the **VX_TASK_INITIALIZE** macro can be used with **VX_TASK** to initialize a task but leave it suspended until it is started later with the **taskActivate( )** function.

The **VX_TASK_INSTANTIATE** returns the task ID if the task is spawned, or **ERROR** if not. The same task name must be used with the **VX_TASK_INSTANTIATE** and **VX_TASK** macros. For example:

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

VX_TASK(myTask,4096);
int myTaskId;

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

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

In order to initialize a task, but keep it suspended until later the **VX_TASK_INITIALIZE** macro can be used. The **taskActivate( )** function must then be used to run the task. The same arguments must be used with both **VX_TASK_INSTANTIATE** and **taskActivate( )**. For example:

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

VX_TASK(myTask,4096);
int myTaskId;

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

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

For more information about these macros and functions see the **taskLib** API reference entry.

For general information about static instantiation, see *1.5 Static Instantiation of Kernel Objects*, p.7.

6.7 Task Names and IDs

A task must be named explicitly (using an ASCII string of any length) if it is created with **taskOpen( )**. It does not have to be named if it is created with **taskSpawn( )**,

When **taskSpawn( )** is used, a null pointer is supplied for the *name* argument results in automatic assignment of a unique name. The name is of the form **t***N*, where *N* is a decimal integer that is incremented by one for each task that is created without an explicit name. Regards of how a task is created, a task ID is returned when it is spawned.

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**.

A task must be created with a name that begins with a back-slash in order to be a public object accessible throughout the system (and not just in the memory context in which it was created—process or kernel). For more information, see *Public Naming of Tasks for Inter-Process Communication*, p.93.

Most VxWorks task functions take a task ID as the argument specifying a task. VxWorks uses a convention that a task ID of 0 (zero) always implies the calling task. In the kernel, the task ID is a 4-byte handle to the task's data structures.

Task Naming Rules

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—all processes and the kernel. For more information about public tasks, see *Public Naming of Tasks for Inter-Process Communication*, p.93.
- 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 function or variable names.

Task Name and ID Functions

The **taskLib** functions listed in Table 6-3 manage task IDs and names.

Table 6-3 **Task Name and ID Functions**

Function	Description
taskName( )	Gets the task name associated with a task ID (restricted to the context—process or kernel—in which it is called).
taskNameToId( )	Looks up the task ID associated with a task name.
taskIdSelf( )	Gets the calling task's ID.
taskIdVerify( )	Verifies the existence of a specified task.

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

Public Naming of Tasks for Inter-Process Communication

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 (the kernel and all processes). Whether tasks are created as private or public objects is determined by how they are named when created. Public tasks must be explicitly named with a forward-slash prefix (for example, **/tMyTask**). For more information about naming tasks, see *6.7 Task Names and IDs*, p. 92.

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( )** function, respectively).

For detailed information, see *7.18 Inter-Process Communication With Public Objects*, p. 147, the **taskOpen** entry in the *VxWorks Kernel API Reference*, and the **taskLib** entry in the *VxWorks Application API Reference*.

6.8 Task Options

When a task is created, you can pass in one or more option parameters. The result is determined by performing a logical OR operation on the specified options.

Table 6-4 **Task Options**

Name	Description
VX_ALTIVEC_TASK	Execute with Altivec coprocessor support (PowerPC only).
VX_SPE_TASK	Execute with SPE support (PowerPC only).
VX_DSP_TASK	Execute with DSP coprocessor support (SuperH only).

Table 6-4 **Task Options** (cont'd)

Name	Description
VX_PRIVATE_ENV	Include private environment support (see the **envLib** API reference). Kernel only.
VX_FP_TASK	Execute with floating-point coprocessor support (see *Floating Point Operations*, p.94).
VX_NO_STACK_FILL	Does not fill the stack with 0xEE (see *Filling Task Stacks*, p.94).
VX_NO_STACK_PROTECT	Create without stack overflow or underflow guard zones (see *6.9 Task Stack*, p.95).
VX_TASK_NOACTIVATE	Used with **taskOpen( )** so that the task is not activated.
VX_UNBREAKABLE	Disables breakpoints for the task; does not allow breakpoint debugging (kernel only).
VX_PRIVATE_CWD	Create with a task-specific (private) current working directory variable (kernel only). Requires configuration of VxWorks with the **PERTASK_CWD** VSB option (see *Kernel Task-Specific Current Working Directory*, p.94).

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 functions perform floating-point operations internally. The VxWorks documentation for each of these functions 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( )** function. 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).

Kernel Task-Specific Current Working Directory

To create a task-specific (private) current working directory variable for a kernel task, use the **VX_PRIVATE_CWD** option when creating the task with **taskOpen( )**, **taskCreate( )**, or **taskSpawn( )**. Without this option, the task shares the kernel's global current working directory variable with all other kernel tasks that have been created without the **VX_PRIVATE_CWD** option. (For RTPs, each RTP has its own current working directory variable that is shared by all tasks in that RTP, which cannot be changed.)

VxWorks must be configured with the **PERTASK_CWD** VSB option (which it is by default) to provide support for task-specific current working directory variables. If VxWorks is not configured the **PERTASK_CWD** VSB option, the task creation option **VX_PRIVATE_CWD** option is ignored.

Getting and Resetting Kernel Task Creation Options

After a kernel task is spawned, you can examine or alter task options by using the functions listed in Table 6-5.

Table 6-5 **Kernel Task Option Functions**

Function	Description
taskOptionsGet( )	Gets task options.
taskOptionsSet( )	Changes task option. The only option that can be changed after a task has been created is **VX_UNBREAKABLE**.

6.9 Task Stack

The size of each task's stack is defined when the task is created. Task stacks can, however, be protected with guard zones and by making task stacks non-executable.

It can be difficult to know exactly how much stack space to allocate. To help avoid stack overflow and corruption, you can initially allocate 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. (For information about setting the initial size, see *6.6 Task Creation and Activation*, p.90).

In addition to experimenting with task stack size, you can also configure and test systems with guard zone protection for task stacks.

CAUTION: For 64-bit VxWorks, due to the larger size of several data types in the LP64 data model, a task is likely to need a larger stack than what is sufficient for 32-bit VxWorks. Typically an increase of 35% to 50% is enough to most applications, although there may be cases in which the stack size must be doubled. Use the **checkStack( )** shell command to determine if your tasks' stacks are too small (that is, the high water mark is too close to the size of the stack).

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.

By default, kernel mode tasks do not have any task stack protection. Configuring VxWorks with the **INCLUDE_PROTECT_TASK_STACK** component provides underflow and overflow guard zones on the *execution stacks,* but none for the *exception stacks*. Stack guard zones in the kernel are mapped to physical memory.

Note that the protection provided for user-mode tasks by configuring the system with the **INCLUDE_RTP** component does not apply to kernel tasks

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 *6.8 Task Options*, p. 93). If a task is created with this option, no guard zones are created for that task.

For RTP tasks, 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.

For kernel tasks, the size of the guard zones are defined by the following configuration parameters:

- **TASK_KERNEL_EXEC_STACK_OVERFLOW_SIZE** for kernel task execution stack overflow size.
- **TASK_KERNEL_EXEC_STACK_UNDERFLOW_SIZE** for kernel task execution stack underflow size.

The value of the parameters for both kernel and RTP tasks 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 user-mode processes (RTPs), 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 *POSIX Thread Stack Guard Zones For RTP Applications*, p.196.

In the kernel, stack guard zones in the kernel consume RAM, as guard zones correspond to mapped memory for which accesses are made invalid.

Non-Executable Kernel Task Stacks

VxWorks creates kernel task stacks with a non-executable attribute when the system is configured with the **INCLUDE_TASK_STACK_NO_EXEC** component, and if the CPU supports making memory non-executable on an MMU-page basis. The size of a stack is always rounded up to a multiple of an MMU page size when the stack is made non-executable (as is also the case when guard zones are inserted).

6.10 Task Information

VxWorks provides functions that get information about a task by taking a snapshot of a task's context when the function 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 Functions**

Function	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.

Table 6-6 **Task Information Functions** (cont'd)

Function	Description
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 *Task-Specific Variables*, p. 107.

6.11 Task Execution Control

Various functions listed provide direct control over a task's execution, allowing for suspending, resuming, restarting and delaying a task.

Table 6-7 **Task Execution Control Functions**

Function	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 function **sysClkRateGet( )** returns the speed of the system clock in ticks per second. Instead of **taskDelay( )**, you can use the POSIX function **nanosleep( )** to specify a delay directly in time units. Only the units are different; the resolution of both delay functions is the same, and depends on the system clock. For details, see *9.11 POSIX Clocks and Timers*, p. 190.

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( )** function treats zero as an error. For information about the ready queue, see *Scheduling and the Ready Queue*, p.87.

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.

6.12 Task Scheduling Control

Tasks are assigned a priority when they are created. Task control functions allow you to change the task priority for both kernel and RTP tasks, and to perform other modifications of scheduling for kernel tasks.

The ability to change task priorities dynamically allows applications to track precedence changes in the real world. You can change a task's priority level while it is executing by calling **taskPrioritySet( )**. Be aware, however, that when 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 assignment of a task's priority when it is created (see *6.6 Task Creation and Activation*, p.90). For information about the ready queue, see *Scheduling and the Ready Queue*, p.87.

Note that for RTPs 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.214.

The full set of functions that can be used to control kernel task scheduling are listed in Table 6-8. Only **taskPrioritySet( )** can be used in RTP tasks.

Table 6-8 **Kernel Task Scheduling Control Functions**

Function	Description
taskPrioritySet( )	Changes the priority of a task.
kernelTimeSlice( )	Controls round-robin scheduling. See *Round-Robin Scheduling*, p.88. Kernel only.
taskRotate( )	Shifts a task from first to last place in its ready queue priority list. See *Task Rotation: Shifting a Kernel Task to End of Priority List*, p.87. Kernel only.

Table 6-8 **Kernel Task Scheduling Control Functions** (cont'd)

Function	Description
taskLock( )	Disables task rescheduling (as long as the task is not blocked or has not voluntarily give up the CPU). See *7.4 Task Locks*, p. 114. Kernel only.
taskUnlock( )	Enables task rescheduling. Kernel only.

6.13 Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system, as well as protected from deletion.

Table 6-9 **Task-Deletion Functions**

Function	Description
taskDelete( )	Terminates a specified task and frees memory—task stacks and task control blocks only. Memory that is allocated by the task during its execution is not freed when the task is terminated. The calling task may terminate itself with this function.
taskUnsafe( )	Undoes a **taskSafe( )**, which makes calling task available for deletion.

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.

Kernel tasks implicitly call **exit( )** if the entry function specified during task creation returns.

An RTP implicitly calls **exit( )**, thus terminating all tasks within it, if the process' **main( )** function returns. For more information see *RTP Termination*, p. 16.

RTP tasks implicitly call **taskExit( )** if the entry function specified during task creation returns.

When a task is deleted, no other task is notified of this deletion. The functions **taskSafe( )** and **taskUnsafe( )** address problems that stem from unexpected deletion of tasks. The function **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 goes here */

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 *7.8 Mutual-Exclusion Semaphores*, p.123.

6.14 Tasking Extensions: Using Hook Functions

VxWorks provides hook management APIs that allow you to register hook functions (hooks) that are then invoked when a task is created or a task is deleted; and additionally for the kernel, when a task context switch occurs.

Spare fields in the task control block (TCB) are used for application extension of a task's context.

The VxWorks task hook management functions are listed in Table 6-10; for more information, see the *VxWorks Application API Reference* for **taskHookLib** and the *VxWorks Kernel API Reference* for **taskCreateHookLib** and **taskSwitchHookLib**.

Table 6-10 **Task Hook Management Functions**

Function	Description
taskCreateHookAdd( )	Adds a function to be called at every task create.
taskCreateHookDelete( )	Deletes a previously added task create function.
taskSwitchHookAdd( )	Adds a function to be called at every task switch. Kernel only.
taskSwitchHookDelete( )	Deletes a previously added task switch function. Kernel only.
taskDeleteHookAdd( )	Adds a function to be called at every task delete.
taskDeleteHookDelete( )	Deletes a previously added task delete function.

Task create hook functions 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 function.

Since create hook functions execute in the context of the creator task, new kernel objects are owned by the creator task's process in the case of RTPs. It may be necessary to assign ownership of these objects to the new task's RTP in order to prevent unexpected object reclamation from occurring if and when the RTP of the creator task terminates.

When the creator task is a kernel task, the kernel owns any kernel objects that are created. There is, therefore, no concern about unexpected object reclamation for this case.

Kernel Usage

When using kernel task switch hook functions, be aware of the following restrictions:

- Do not assume any virtual memory (VM) context is current other than the kernel critical section (as with ISRs).
- Do not rely on knowledge of the current task or invoke any function that relies on this information, for example **taskIdSelf( )**.
- Do not rely on **taskIdVerify (pOldTcb)** to determine if a delete hook has executed for the self-destructing task case. Instead, other state information must be changed in the delete hook to be detected by the switch hook (for example by setting a pointer to NULL).

User-installed switch hooks are called within the kernel critical section (where tasks states are manipulated), and therefore do not have access to all VxWorks facilities. Table 6-11 summarizes the functions that can be called from a task switch hook; in general, any function that does not involve a kernel section can be called.

Table 6-11 **Functions Callable by Kernel Task Switch Hooks**

Library	Functions
bLib	All functions
fppArchLib	**fppSave( )**, **fppRestore( )**
intLib	**intContext( )**, **intCount( )**, **intVecSet( )**, **intVecGet( )**, **intLock( )**, **intUnlock( )**
lstLib	All functions except **lstFree( )**
mathALib	All are callable if **fppSave( )**/**fppRestore( )** are used
rngLib	All functions except **rngCreate( )**
taskLib	**taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**, **taskIsSuspended( )**, **taskIsPended( )**, **taskIsDelayed( )**, **taskTcb( )**
spinlocklib	**spinLockIsrTake( )** and **spinLockIsrGive( )**
spinLockIsrNdLib	**spinLockIsrNdTake( )** and **spinLockIsrNdGive( )**

Table 6-11 **Functions Callable by Kernel Task Switch Hooks** (cont'd)

Library	Functions
vxCpuLib	**vxCpuIndexGet( )**
vxLib	**vxTas( )**

NOTE: For information about POSIX extensions, see *9. POSIX Facilities*.

6.15 Task Error Status: errno

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

NOTE: This section describes the implementation and use of **errno** in UP configurations of VxWorks, which is different from that in SMP configurations. For information about **errno** and other global variables in VxWorks SMP, see *SMP CPU-Specific Variables and Uniprocessor Global Variables*, p.456. For information about migration, see *18.18 Code Migration for VxWorks SMP*, p.449.

Layered Definitions of errno

In the VxWorks kernel, **errno** is simultaneously defined in two different ways. There is, as in ANSI C, an underlying global variable called **errno**, which you can display by name using host development tools.

However, **errno** is also defined as a macro in **errno.h**; this is the definition visible to all of VxWorks except for one function. The macro is defined as a call to a function **__errno( )**that returns the address of the global variable, **errno** (as you might guess, this is the single function that does not itself use the macro definition for **errno**). This subterfuge yields a useful feature: because **__errno( )**is a function, you can place breakpoints on it while debugging, to determine where a particular error occurs.

Nevertheless, because the result of the macro **errno** is the address of the global variable **errno**, C programs can set the value of **errno** in the standard way:

```
errno = someErrorNumber;
```

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

A Separate errno Value for Each Task

In RTPs (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).

In the VxWorks kernel, the underlying global **errno** is a single predefined global variable that can be referenced directly by application code that is linked with VxWorks (either statically on the host or dynamically at load time).

However, for **errno** to be useful in the multitasking environment of VxWorks, each task must see its own version of **errno**. Therefore **errno** is saved and restored by the kernel as part of each task's context every time a context switch occurs.

Similarly, *interrupt service routines (ISRs)* see their own versions of **errno**. This is accomplished by saving and restoring **errno** on the interrupt stack as part of the interrupt enter and exit code provided automatically by the kernel (see *8.12 ISR Connection to Interrupts*, p.171).

Thus, regardless of the VxWorks context, an error code can be stored or consulted without direct manipulation of the global variable **errno**.

Error Return Convention

Almost all VxWorks functions follow a convention that indicates simple success or failure of their operation by the actual return value of the function. Many functions return only the status values **OK** (0) or **ERROR** (-1). Some functions that normally return a nonnegative number (for example, **open( )** returns a file descriptor) also return **ERROR** to indicate an error. Functions that return a pointer usually return **NULL** (0) to indicate an error. In most cases, a function returning such an error indication also sets **errno** to the specific error code.

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

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

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

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

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 **errnoLib** API reference for more information about defining and decoding **errno** values with this convention.

6.16 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. These exceptions can be handled by the VxWorks exception handling package or with the signal facility.

For information about the VxWorks exception handling package see *13. 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.

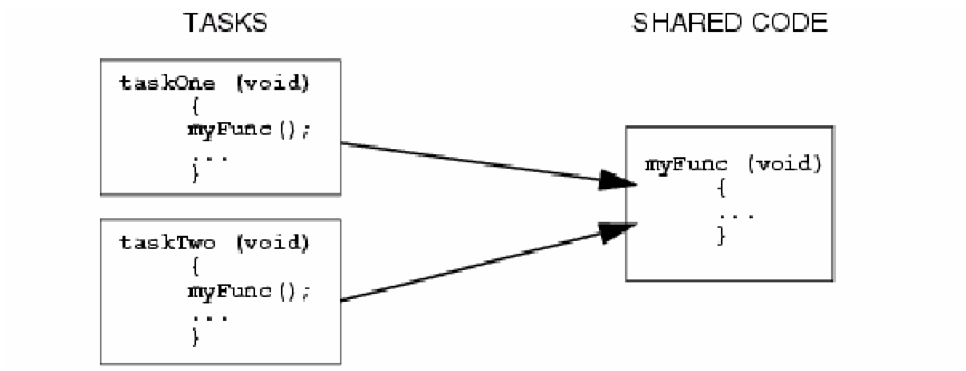
Signals are also used for signaling software exceptions as well as hardware exceptions. They are described in more detail in *8.1 About Signals*, p.152 and in the API reference for **sigLib**.

6.17 Shared Code and Reentrancy

In VxWorks, it is common for a single copy of a function or library to be invoked by many different tasks. For example, many tasks may call **printf( )**, but there is only a single copy of the function in the system. A single copy of code executed by multiple tasks is called *shared code*. Shared code must be re-entrant.

VxWorks dynamic linking facilities make sharing code especially easy. Shared code makes a system more efficient and easier to maintain.

Figure 6-4 **Shared Code**


TASKS
SHARED CODE
taskOne (void) { myFunc(); ... }
taskTwo (void) { myFunc(); ... }
myFunc (void) { ... }

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

Most functions in VxWorks are reentrant. However, you should assume that any function *someName*( ) is not reentrant if there is a corresponding function named *someName*_r( ) — the latter is provided as a reentrant version of the function. For example, because **ldiv( )** has a corresponding function **ldiv_r( )**, you can assume that **ldiv( )** is not reentrant.

VxWorks kernel I/O and driver functions are reentrant, but require careful application design. For buffered I/O, Wind River recommends using file-pointer buffers on a per-task basis. At the driver level, it is possible to load buffers with streams from different tasks, due to the global file descriptor table in VxWorks.

This may or may not be desirable, depending on the nature of the application. For example, a packet driver can mix streams from different tasks because the packet header identifies the destination of each packet.

The majority of VxWorks functions use the following reentrancy techniques:

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

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

NOTE: In some cases reentrant kernel code is not preferable. A critical section should use a binary semaphore to guard it, or use **intLock( )** or **intUnlock( )** if called from by an ISR.

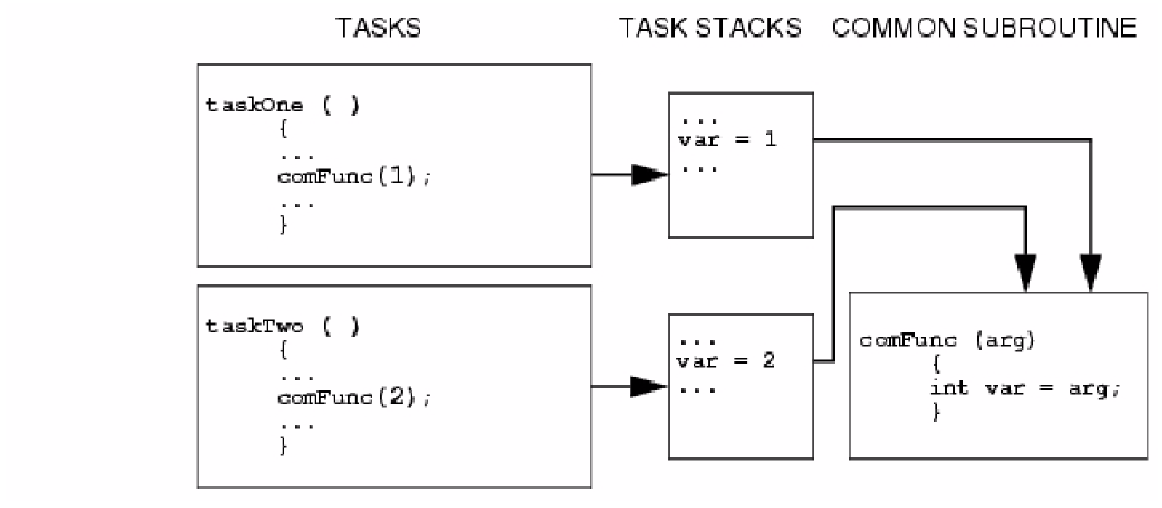
NOTE: Initialization functions should be callable multiple times, even if logically they should only be called once. As a rule, functions should avoid **static** variables that keep state information. Initialization functions are an exception; using a **static** variable that returns the success or failure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

Dynamic Stack Variables and Re-entrancy

Many functions 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 functions operate on lists and nodes provided by the caller in each function call.

Functions of this kind are inherently reentrant. Multiple tasks can use such functions 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**



Guarded Global and Static Variables

Some libraries encapsulate access to common data. This kind of library requires some caution because the functions are not inherently reentrant. Multiple tasks simultaneously invoking the functions 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 *7.8 Mutual-Exclusion Semaphores*, p.123.

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.

Also note that each task has a VxWorks events register, which receives events sent from other tasks, ISRs, semaphores, or message queues. See *7.16 VxWorks Events*, p.138 for more information about this register, and the functions used to interact with it.

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 *18.18 Code Migration for VxWorks SMP*, p.449.

Thread-Local Variables: __thread Storage Class

Thread-local (**__thread**) 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).

Configure VxWorks with the **INLCUDE_TLS** component for thread-local storage support.

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.

For more information see the **tlsLib** API reference.

CAUTION: Do not access **__thread** variables from an ISR. Doing so may have unpredictable results.

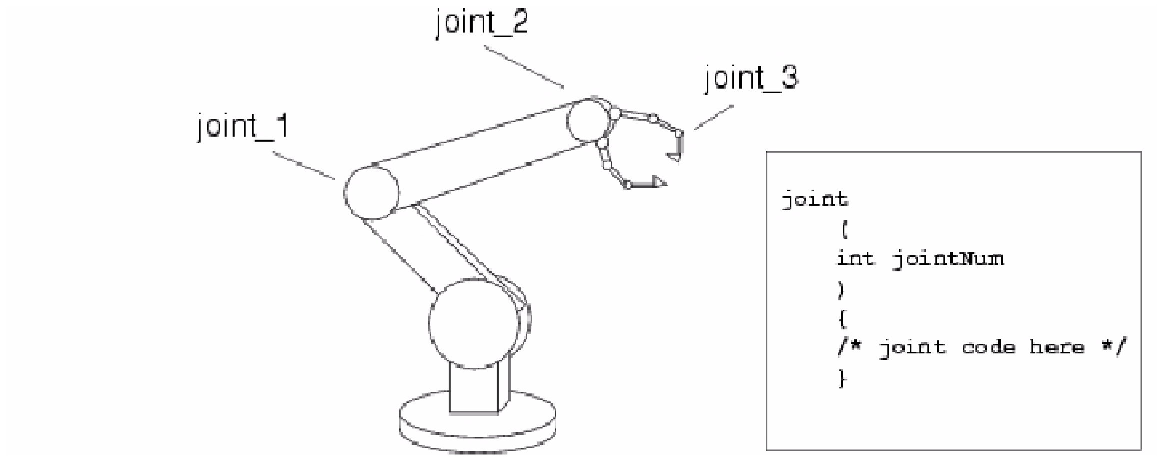
Multiple Tasks with the Same Main Function

With VxWorks, it is possible to spawn several tasks with the same main function. Each spawn creates a new task with its own stack and context. Each spawn can also pass the main function different parameters to the new task. In this case, the same rules of reentrancy described in *Task-Specific Variables*, p.107 apply to the entire task.

This is useful when the same function must be performed concurrently with different sets of parameters. For example, a function 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 function 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 */
    }
```

6.18 Kernel Task Environment Variables

By default kernel tasks share a common set of environment variables. But a task has its own private environment if it is created with the **VX_PRIVATE_ENV** task creation option.

For information about task creation options, see *6.8 Task Options*, p.93.

envLib Environment Variable APIs

The **envLib** library provides a Unix-compatible environment variable facility, with the following functions:

Function	Description
envLibInit( )	Initialize environment variable facility.
envPrivateCreate( )	Create a private environment.
envPrivateDestroy( )	Destroy a private environment.
putenv( )	Set an environment variable.
getenv( )	Get an environment variable (ANSI).
getenv_s( )	Get an environment variable.
envShow( )	Display the environment for a task.
envGet( )	Return a pointer to the environment of a task.

For more information, see the **envLib** entry in the *VxWorks 7 Kernel API Reference*.

7

Intertask and Interprocess Communication

7.1 About Intertask and Interprocess Communication

VxWorks intertask and interprocess facilities provide mechanisms for synchronization of the activity of different tasks as well as for communication between them.

For interprocess and kernel-process communication, VxWorks semaphores and message queues, pipes, and events (as well as POSIX semaphores and events) can be created as *public* objects to provide for intertask communication across memory boundaries (between the kernel and processes, and between different processes).

VxWorks provides the following facilities for intertask and interprocess communication:

Shared data structures
: Provide a means for tasks executing in the same memory space (kernel or RTP) to communicate by accessing data structures. See *7.2 Shared Data Structures*, p. 113.

Interrupt locks (kernel only)
: Provide a means for disabling interrupts, and thus preventing preemption by ISRs. Interrupt locking is not a general purpose mechanism, and should only be used in a very restricted manner. See *7.3 Interrupt Locks*, p. 113.

Task locks (kernel only)
: Provide a means for disabling preemption by other tasks. Task locking is not a general purpose mechanism, and should only be used in a very restricted manner. See *7.4 Task Locks*, p. 114.

Semaphores
: Provide the primary means for synchronization of tasks and mutual exclusion. General purpose and specialized variants are available, as described in *7.5 Types of Semaphores*, p. 115. Semaphores can be created as public objects, which allows for their use in interprocess communication; for information in this regard, see *7.18 Inter-Process Communication With Public Objects*, p. 147. For information about POSIX semaphores, see *9.19 POSIX Semaphores*, p. 217.

Message queues
: Provide a higher-level mechanism for direct communication of messages between tasks. See *7.12 Message Queues*, p. 131. Message queues can be created as public objects, which allows for their use in interprocess communication; for information in this regard, see *7.18 Inter-Process Communication With Public Objects*, p. 147.

Pipes
: Provide an alternative messaging interface to the message queue facility. Pipes operate through the I/O system, which allows for use of standard I/O functions and **select( )**. See *7.14 Pipes*, p. 136.

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. See *7.16 VxWorks Events*, p. 138.

VxWorks condition variables
: Provide a fundamental task synchronization capability that combines thread-safe access to a resource with event signaling capabilities. See *7.17 VxWorks Condition Variables*, p. 144.

For information about signals, which are not intended to be used for general-purpose intertask and interprocess communication, see *8.1 About Signals*, p. 152.

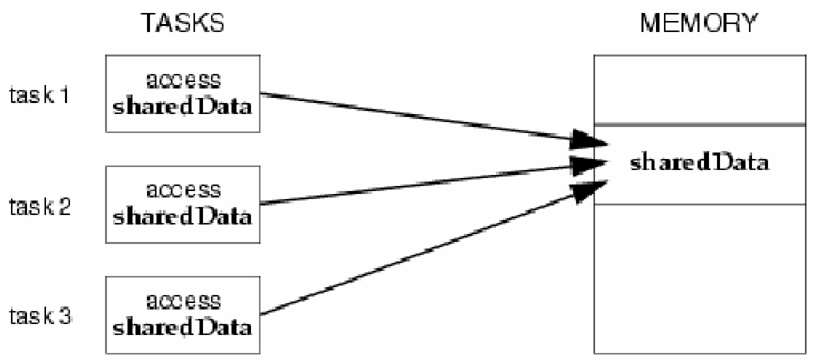
NOTE: With few exceptions, the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks share the same facilities for intertask and interprocess communications—the difference amounts to only a few functions. For information about the VxWorks SMP, see *18. VxWorks SMP*; and for information specifically about migration, see *18.18 Code Migration for VxWorks SMP*, p. 449.

7.2 Shared Data Structures

The most obvious way for tasks executing in the same memory space (within a single 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.

Figure 7-1 **Shared Data Structures**



Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced directly by code running in different task contexts. Access to shared data structures should, however, be controlled using a mutual exclusion such as semaphores (see *7.5 Types of Semaphores*, p. 115).

7.3 Interrupt Locks

In the kernel, the **intLock( )** function can be used to disable interrupts, and thus prevents preemption by ISRs. It can be called from a task or ISR context. The **intUnLock( )** function reenables interrupts.

The functions are used to protect a critical region of code as follow:

```
foo ()
    {
    int lockKey = intLock();
    .
    .   /* critical region of code that cannot be interrupted */
    .
    intUnlock(lockKey);
    }
```

When a task is accessing a variable or data structure that is also accessed by an ISR, use **intLock( )** to prevent preemption. The operation should be minimal, meaning a few lines of code and no function calls. If the call is too long, it can directly impact interrupt latency and cause the system to become far less deterministic.

To prevent preemption of a task by both tasks and ISRs, use both **intLock( )** and **taskLock( )**; for more information in this regard, see the API reference entry for **intLock( )**.

For information about ISRs, see *8.7 About Interrupt Service Routines: ISRs*, p.163.

WARNING: Invoking a VxWorks system function with interrupts locked may result in interrupts being re-enabled for an unspecified period of time. If the called function blocks, or results in a higher priority task becoming eligible for execution (**READY**), interrupts will be re-enabled while another task executes, or while the kernel is idle. For more information in this regard, see the API reference entry for **intLock( )**.

NOTE: The **intLock( )** function is provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including the ISR-callable spinlock, which defaults to **intLock( )** behavior in a UP system. For more information, see *ISR-Callable Spinlocks*, p.423 and *18.18 Code Migration for VxWorks SMP*, p.449.

7.4 Task Locks

In the kernel, the **taskLock( )** function disables preemption of the calling task by other tasks.

The calling task is the only task that is allowed to execute—as long as it is in the ready state. If the calling task blocks or suspends, the scheduler selects the next highest-priority eligible task to execute. When the calling task unblocks and resumes execution, preemption is again disabled. (For information about task states, see *6.4 Task States and Transitions*, p.82.) The **taskUnlock( )** function reenables preemption by other tasks.

The task lock functions are used to protect a critical region of code as follow:

```
foo()
    {
    taskLock();
    .
    .  /* critical region of code that cannot be interrupted */
    .
    taskUnlock();
    }
```

Task locks can be nested (they are implemented using a count variable), in which case preemption is not reenabled until **taskUnlock( )** has been called as many times as **taskLock( )**.

Task locks can lead to unacceptable real-time response times. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region. While this kind of mutual exclusion is simple, be sure to keep the duration short. In general, semaphores provide a better mechanism for mutual exclusion; see *7.5 Types of Semaphores*, p.115.

Interrupts are not blocked when **taskLock( )** is used, but it can be paired with **intLock( )** to disable preemption by tasks and ISRs.

For more information, see the API reference entries for **taskLock( )** and **taskUnLock( )**.

NOTE: The **taskLock( )** and **taskUnlock( )** functions are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternatives are available for SMP systems, including task-only spinlocks, which default to **taskLock( )** and **taskUnlock( )** behavior in a UP system. For more information, see *Task-Only Spinlocks*, p.425 and *18.18 Code Migration for VxWorks SMP*, p.449.

7.5 Types of 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.

For *mutual exclusion*, semaphores interlock access to shared resources. In the kernel they provide mutual exclusion with finer granularity than either locking interrupts or task (see *7.3 Interrupt Locks*, p.113 and *7.4 Task Locks*, p.114).

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 *18.7 Memory Barriers*, p.429.

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 *7.7 Binary Semaphores*, p.120.

mutual exclusion
: A special binary semaphore optimized for problems inherent in mutual exclusion: priority inversion, deletion safety, and recursion. For more information, see *7.8 Mutual-Exclusion Semaphores*, p.123.

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 *7.9 Counting Semaphores*, p.128.

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 *7.10 Read/Write Semaphores*, p.128.

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 *9.19 POSIX Semaphores*, p.217.

7.6 Semaphore Creation and Use

In most cases, VxWorks provides a single, uniform interface for semaphore control—instead of defining a full set of semaphore control functions specific to each type of semaphore.

The exceptions to this rule are as follows:

- The creation functions, which are specific to each semaphore type.
- The give and take functions for read/write semaphores, which support read and write modes for each operation.
- The scalable and inline variants of the give and take kernel functions—for binary and mutex semaphores—which provide optimized alternatives to the standard functions. See *Scalable and Inline Semaphore Take and Give Kernel Functions*, p.119.

Semaphore Control Functions

Table 7-1 **Semaphore Control Functions**

Function	Description
semBInitialize( )	Initializes a pre-allocated binary semaphore. See *Static Instantiation of Kernel Semaphores*, p.118. Kernel only.
semCInitialize( )	Initializes a pre-allocated counting semaphore. See *Static Instantiation of Kernel Semaphores*, p.118. Kernel only.
semMInitialize( )	Initializes a pre-allocated mutual-exclusion semaphore. See *Static Instantiation of Kernel Semaphores*, p.118. Kernel only.
semRWInitialize( )	Initializes a pre-allocated read/write semaphore. See *Static Instantiation of Kernel Semaphores*, p.118. Kernel only.

Table 7-1 **Semaphore Control Functions** (cont'd)

Function	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.
semRTake( )	Takes a read/write semaphore in read mode.
semWTake( )	Takes a read/write semaphore in write mode.
semGive( )	Gives a binary, mutual-exclusion, or counting semaphore.
semRWGive( )	Gives a read/write semaphore.
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.
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 functions return a semaphore ID that serves as a handle on the semaphore during subsequent use by the other semaphore-control functions. 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.

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. To create a semaphore as a public object, the **semOpen( )** function must be used and the semaphore name must begin with forward-slash (for example **/mySem**). The type of semaphore is specified with the semaphore-type argument.

For detailed information, see *7.18 Inter-Process Communication With Public Objects*, p. 147, the **semOpen** entry in the *VxWorks Kernel API Reference*, and the **semLib** entry in the *VxWorks Application API Reference*.

Options for Scalable and Inline Semaphore Kernel Functions

Table 7-2 **Scalable and Inline Kernel Semaphore Options**

Function	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.

For general information about scalable and inline functions, see *Scalable and Inline Semaphore Take and Give Kernel Functions*, p. 119.

⚠ **CAUTION:** The options listed in Table 7-2 must not be used when semaphores are created. Errors are generated if they are used.

Static Instantiation of Kernel Semaphores

The dynamic semaphore creation functions perform a dynamic, two-step operation, in which memory is allocated for the semaphore object at runtime, and then the object is initialized. Semaphores (and other VxWorks objects) can also be statically instantiated—which means that their memory is allocated for the object at compile time—and the object is then initialized at runtime.

The **VX_BINARY_SEMAPHORE**, **VX_COUNTING_SEMAPHORE**, **VX_MUTEX_SEMAPHORE**, and **VX_READ_WRITE_SEMAPHORE** macros declare a semaphore of type binary, counting, and mutex (respectively) at compile time. These macros take the semaphore name as an argument.

The semaphores declared by these macros are initialized by calling functions **semBInitialize( )**, **semCInitialize( ) semMInitialize( )**, and **semRWInitialize( )** respectively. The three **sem*X*Initialize( )** functions are the equivalents of their respective **sem*X*Create( )** functions.

The same semaphore names must be used with the pair of **VX_*XXX*_SEMAPHORE** macro and **sem*X*Initialize( )** function. The return value from the **sem*X*Initialize( )** functions is a semaphore ID, which is then used to perform all operations on the semaphores. For example:

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

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

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

For more information, see the API references for **semBLib**, **semCLib**, and **semMLib**.

For general information about static instantiation, see *1.5 Static Instantiation of Kernel Objects*, p.7.

Scalable and Inline Semaphore Take and Give Kernel Functions

In addition to the standard semaphore give and take functions, VxWorks provides scalable and inline of variants for use with binary and mutex semaphores in the kernel. These functions 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 functions even if just inline variants are used, because they provide optimizations for uncontested take and give operations in an SMP system.

The scalable functions are designed for use with lightly contested resources when performance is of greater significance than features of the standard functions 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 functions provide the same options as the scalable functions, 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 function for which there is an inline variant, either a wrapper should be created for the inline function, or the scalable variant should be used instead.

The scalable and inline of variants are listed in Table 7-1; and the options for the scalable and inline functions are listed in Table 7-2). Note that in order to use these functions, you must include the **h/inline/semLibInline.h** header file.

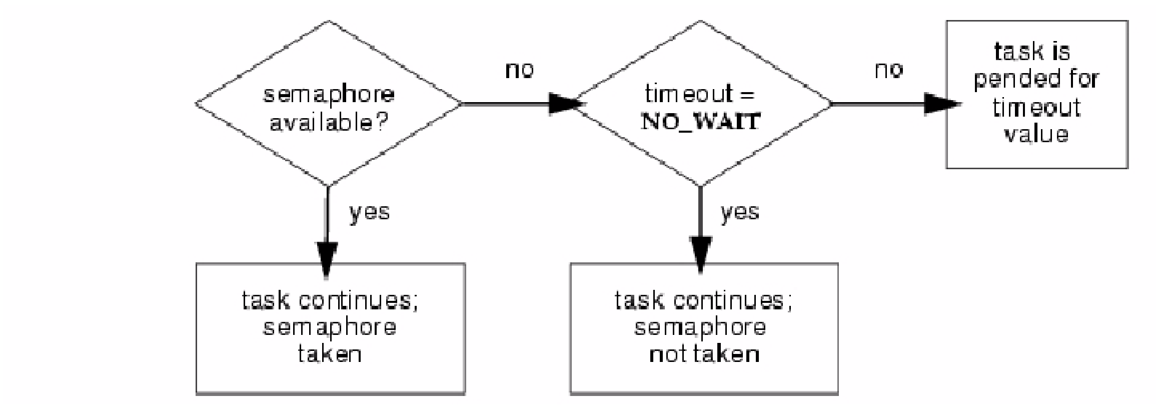
7.7 Binary Semaphores

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 *7.8 Mutual-Exclusion Semaphores*, p. 123 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.

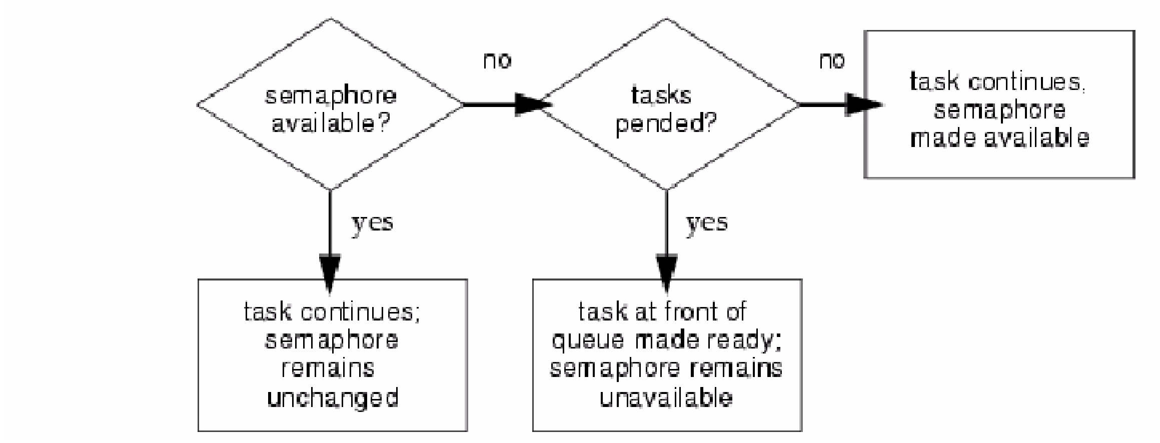
A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with **semTake( )**, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 7-2. If the semaphore is available (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 7-2 **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 7-3. 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 7-3 **Giving a Semaphore**


semaphore available?
no
tasks pended?
no
task continues, semaphore made available
yes
yes
task continues; semaphore remains unchanged
task at front of queue made ready; semaphore remains unavailable

Mutual Exclusion

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

```
/* includes */
#include <vxWorks.h>
#include <semLib.h>

SEM_ID semMutex;

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

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

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

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

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

Synchronization

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

(See *8.7 About Interrupt Service Routines: ISRs*, p.163 for a complete discussion of ISRs)

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

In Example 7-1, the kernel function **init( )** creates the binary semaphore, attaches an ISR to an event, and spawns a task to process the event. The function **task1( )** runs until it calls **semTake( )**. It remains blocked at that point until an event causes the ISR to call **semGive( )**. When the ISR completes, **task1( )** executes to process the event. There is an advantage of handling event processing within the context of a dedicated task: less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

Example 7-1 **Using Semaphores for Kernel Task Synchronization**

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

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

SEM_ID syncSem;             /* ID of sync semaphore */

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

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

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

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

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

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

7.8 Mutual-Exclusion Semaphores

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

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

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

User-Level Mutex Semaphores for RTP Applications

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.

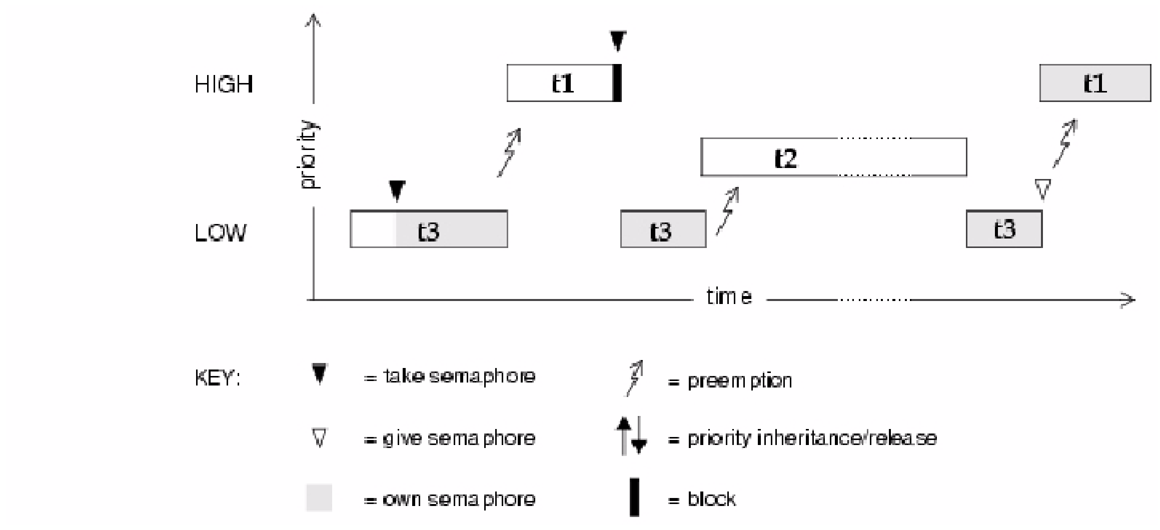
In the case when a mutex semaphore is contested, it will be slower than a kernel-level semaphore because a system call must be made.

By default, using the **semMCreate()** function 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()** function 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.

Priority Inversion and Priority Inheritance

Figure 7-4 illustrates a situation called priority inversion.

Figure 7-4 **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 7-4: **t1**, **t2**, and **t3** are tasks of high, medium, and low priority, respectively. Task **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.

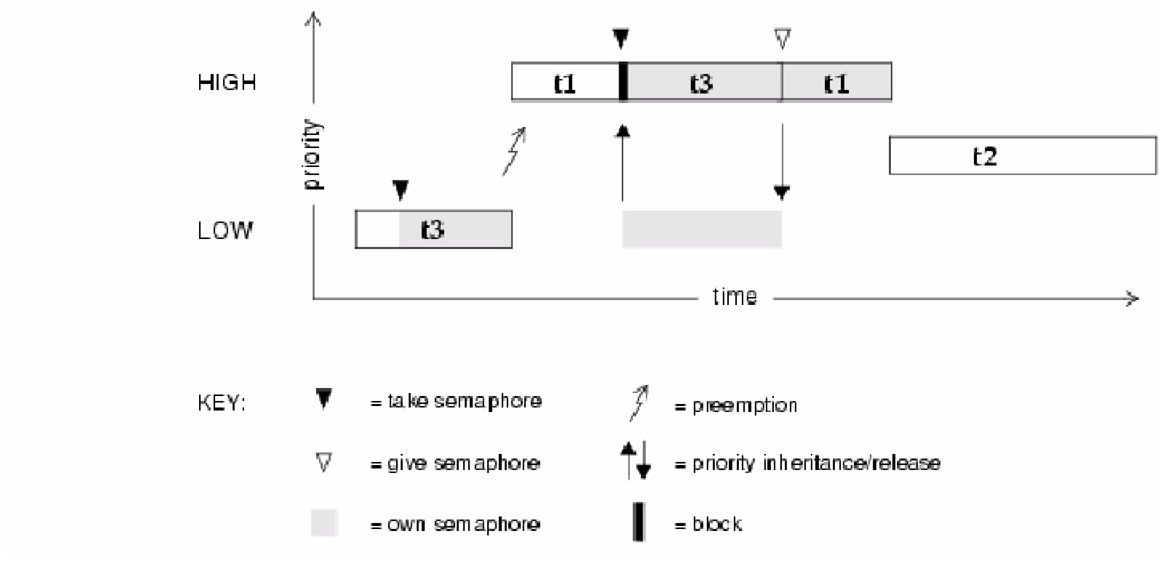
Priority Inheritance Policy

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.87.)

Figure 7-5 **Priority Inheritance**



In Figure 7-5, 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);
```

Priority Inheritance With Multiple Mutual Exclusion Semaphores

For situations in which a task owns more than one mutual-exclusion semaphore at a time (which have priority-inheritance option enabled), the task executes at the highest priority that it has inherited from the tasks that are blocked on any of the mutual-exclusion semaphores that it owns. In addition, it continues to run at that priority until it has relinquished *all* of the semaphores on which the other tasks are blocked.

Figure 7-6 **Priority Inheritance With Multiple Mutex Semaphores**

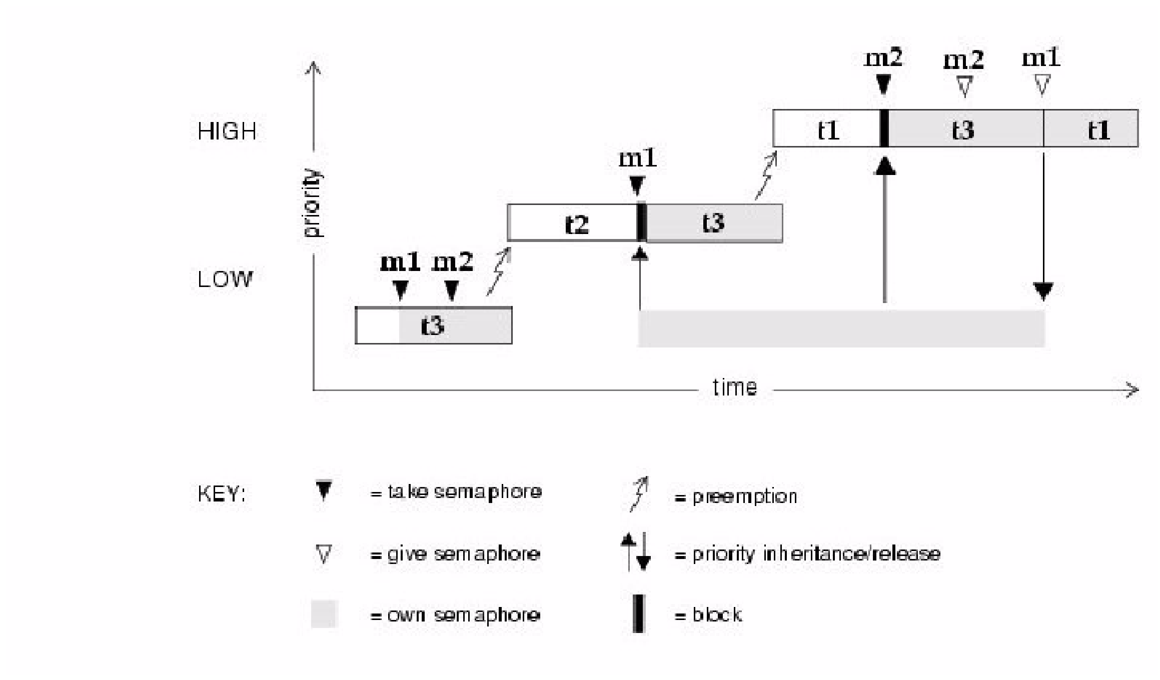


Figure 7-6 illustrates the following example of the interaction of multiple tasks with multiple mutex semaphores:

1. Task **t3** with normal priority 100 takes mutex semaphore **m1**.
2. Task **t3** takes semaphore **m2**.
3. Task **t2** with priority 90 preempts task **t3**.
4. Task **t2** attempts to take semaphore **m1**, and blocks.
5. Task **t3** executes with priority of 90, inherited from task **t2**.
6. Task **t1** with priority 80 preempts task **t3**.
7. Task **t1** attempts to take semaphore **m2**, and blocks.
8. Task **t3** executes with priority 80, inherited from task **t1**.
9. Task **t3** relinquishes semaphore **m2**, remains at priority 80.
10. Task **t3** relinquishes semaphore **m1**, and returns to its normal priority 100.
11. Task **t1** gets **m2**, preempts task **t3**.

While it may not seem optimal to allow task **t3** to run at the priority it inherited from t1 (90) until it has relinquished both semaphores, the VxWorks priority-inheritance policy takes this approach in order to keep semaphore operations fast, with deterministic execution times.

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 functions 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 7-2 **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");
    }
```

7.9 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 7-3 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Table 7-3 **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( )** function.

7.10 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.

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.

Read/write semaphores are particularly suited to SMP systems (for information about the SMP configuration of VxWorks, see *18. VxWorks SMP*.

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 7-1 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 function 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 function 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.

7.11 Special Semaphore Options

For kernel tasks, the uniform VxWorks semaphore interface includes three special options: timeout, queueing, and use with VxWorks events. For RTP tasks, it additionally provides a fourth option: interruptible by signals.

These options are not available for either the read/write semaphores described in *7.10 Read/Write Semaphores*, p.128, or the POSIX-compliant semaphores described in *9.19 POSIX Semaphores*, p.217.

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 functions 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 function returns **OK**. The **errno** set when a take function returns **ERROR** due to timing out before successfully taking the semaphore depends upon the timeout value passed.

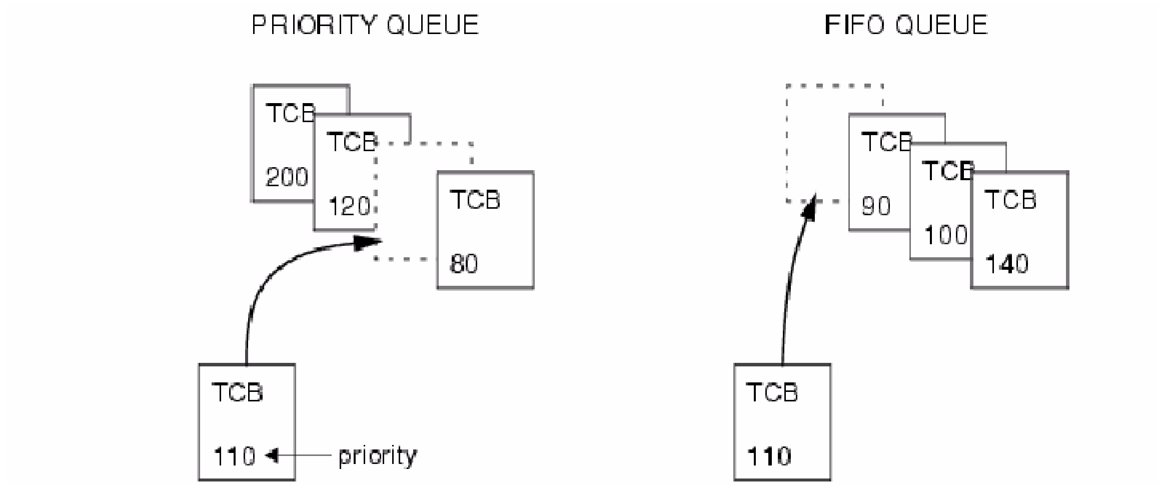
A **semTake( )** with **NO_WAIT** (0), which means *do not wait at all*, sets **errno** to **S_objLib_OBJ_UNAVAILABLE**. A **semTake( )** with a positive timeout value returns **S_objLib_OBJ_TIMEOUT**. A timeout value of **WAIT_FOREVER** (-1) means *wait indefinitely*.

For more information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p.149.

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 7-7.

Figure 7-7 **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 function. Semaphores using the priority inheritance option (**SEM_INVERSION_SAFE**) must select priority-order queuing.

Semaphores Interruptible by Signals in RTPs

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

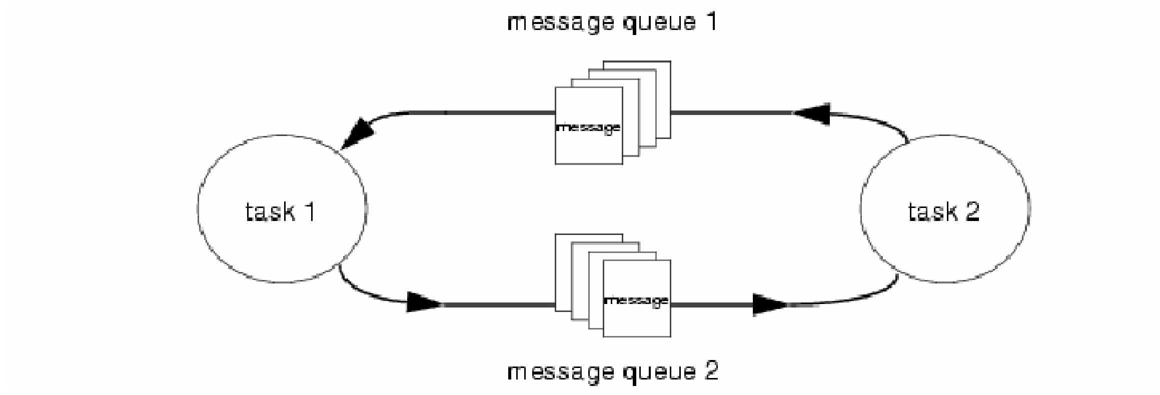
Semaphores can send VxWorks events to a specified task when they becomes free. For more information, see *7.16 VxWorks Events*, p.138.

7.12 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.*

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 7-8 **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 7-8.

There are two message-queue 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 *Comparison of POSIX and VxWorks Semaphores*, p.218 for a discussion of the differences between the two message-queue designs.

7.13 Message Creation and Use

VxWorks message queues are created, used, and deleted with the functions provided by the **msgQLib** library.

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 7-4 **VxWorks Message Queue Control**

Function	Description
msgQInitialize( )	Initializes a pre-allocated message queue. See *Static Instantiation of Kernel Message Queues*, p.133. Kernel only.
msgQCreate( )	Allocates and initializes a message queue.
msgQDelete( )	Terminates and frees a message queue.

Table 7-4 **VxWorks Message Queue Control** (cont'd)

Function	Description
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.

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. To create a message queue as a public object, the **msgQOpen( )** function must be used and the message queue name must begin with forward-slash (for example **/myMsgQ**).

For detailed information, see *7.18 Inter-Process Communication With Public Objects*, p.147, the **msgQOpen** entry in the *VxWorks Kernel API Reference*, and the **msgQLib** entry in the *VxWorks Application API Reference*.

Static Instantiation of Kernel Message Queues

The kernel function **msgQCreate( )** performs a dynamic, two-step operation, in which memory is allocated for the message queue object at runtime, and then the object is initialized. Message queues (and other VxWorks objects) can also be statically instantiated—which means that their memory is allocated for the object at compile time—and the object is then initialized at runtime.

The macro **VX_MSG_Q** declares a message queue at compile time. It takes three parameters: the name, the maximum number of messages in the message queue, and the maximum size of each message. The **msgQInitialize( )** function is used at runtime to initialize the message queue and make it ready for use. The same message queue name—as well as the same values for message queue size and maximum number of messages—must be used with the macro and function. For example:

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

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

STATUS initializeFunction (void)
```

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

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

For general information about static instantiation, see *1.5 Static Instantiation of Kernel Objects*, p.7.

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 function.

For more information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p.149.

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.

Message Queue Code Example

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 in RTPs

By default in an RTP, 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( )** functions) the queuing mechanism that should be used for tasks that pend on **msgQSend( )** and **msgQReceive( )**.

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):

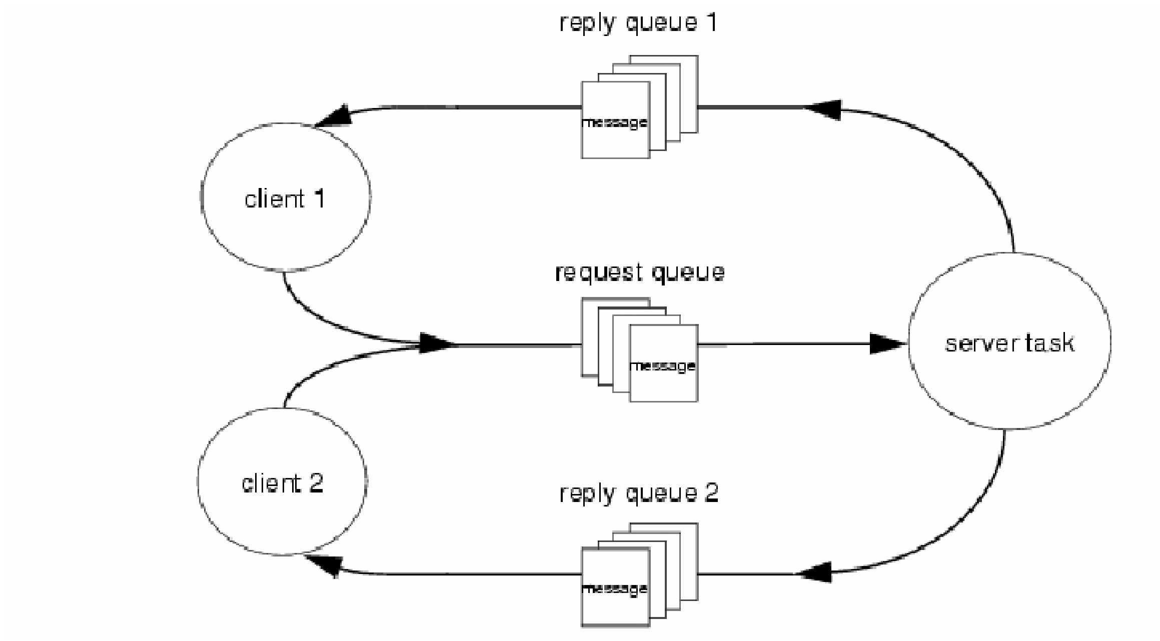
```
-> show myMsgQId
Message Queue Id  : 0x3adaf0
Task Queuing      : FIFO
Message Byte Len  : 4
Messages Max      : 30
Messages Queued   : 14
Receivers Blocked : 0
Send timeouts     : 0
Receive timeouts    : 0
```

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 *7.14 Pipes*, p.136) are a natural way to implement this functionality.

For example, client-server communications might be implemented as shown in Figure 7-9. 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 7-9 **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.

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 *7.16 VxWorks Events*, p.138.

7.14 Pipes

Pipes provide an alternative interface to the message queue facility that goes through the VxWorks I/O system. Pipe devices are managed by the **pipeDrv** virtual I/O device, and use the kernel message queue facility to bear the actual message traffic. Tasks write messages to pipes, which are then read by other tasks. This allows you to implement a client-server model of intertask communications.

As I/O devices, pipes also provide an important feature that message queues themselves cannot—the use of **select( )**. This function allows a task to wait for data to be available on any of a set of I/O devices. The **select( )** function 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 *Pending on Multiple File Descriptors with select( )*, p.304.

Named pipes can be created in RTPs. However, unless they are specifically deleted by the application they will persist beyond the life of the RTP 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.

For more information about pipes, see the **ioLib** entry in the *VxWorks Application API Reference*, and the **pipeDrv** entry in the *VxWorks Kernel API Reference*.

7.15 Pipe Creation and Use

The function **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.

The syntax is as follows:

```
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 functions to open, read, and write pipes, and invoke *ioctl* functions. 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.

Writing to Pipes from ISRs in the Kernel

In the kernel, VxWorks pipes are designed to allow ISRs to write to pipes in the same way as task-level code. Many VxWorks facilities cannot be used from ISRs, including output to devices other than pipes. However, ISRs can use pipes to communicate with tasks, which can then invoke such facilities. ISRs write to a pipe using the **write( )** call. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message is discarded because the ISRs cannot pend. ISRs must not invoke any I/O function on pipes other than **write( )**. For more information ISRs, see *8.7 About Interrupt Service Routines: ISRs*, p.163.

I/O Control Functions for Pipes

Pipe devices respond to the **ioctl( )** functions summarized in Table 7-5. The functions listed are defined in the header file **ioLib.h**. For more information, see the reference entries for **pipeDrv** and for **ioctl( )** in **ioLib**.

Table 7-5 **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.

Table 7-5 **I/O Control Functions Supported by pipeDrv** (cont'd)

Function	Description
FIONMSGS	Gets the number of messages remaining in the pipe.
FIONREAD	Gets the size in bytes of the first message in the pipe.

7.16 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. Events can be sent explicitly by tasks and ISRs, and can be sent when message queues or semaphores are free. Only tasks can receive events.

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 explicitly by other tasks and ISRs, or when semaphores, and message queues are free. 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.

NOTE: VxWorks events, which are also simply referred to as *events* in this section, should not be confused with System Viewer events.

VxWorks Configuration for Events

To provide events facilities, VxWorks must be configured with the **INCLUDE_VXEVENTS** component.

The **VXEVENTS_OPTIONS** configuration parameter controls behavior of RTP tasks with regard to events and signals. It has the following options:

EVENTS_NONINTERRUPTIBLE
Signals do not unpend RTP tasks that are pended on events (the default option).

EVENTS_INTERRUPTIBLE
Signals unpend RTP tasks that are pended on events.

About Event Flags and the Task Events Register

Each task has 32 event flags, bit-wise encoded in a 32-bit word (bits 25 to 32 are reserved for Wind River use), which is referred to as an *event register.* The event register is used to store the events that the task receives from tasks, ISRs, semaphores, and message queues.

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 a specific bit being set. There is no mechanism for recording how many times any given event has been received by a task. Once a flag has been set, its being set again by the same or a different sender is essentially an *invisible* operation.

A a task cannot access the contents of its events registry directly; the **eventReceive( )** function copies the contents of the events register to the variable specified by its ***pEventsReceived** parameter.

The VxWorks event flag macros and values that can be used by Wind River customers are listed in *Table 7-6VxWorks Event Flags*, p.139.

⚠ **CAUTION:** Event flags **VXEV25** (0x01000000) through **VXEV32** (0x80000000) are reserved for Wind River use only, and should not be used by customers.

Table 7-6 **VxWorks Event Flags**

Macro	Value
VXEV01	0x00000001
VXEV02	0x00000002
VXEV03	0x00000004
VXEV04	0x00000008
VXEV05	0x00000010
VXEV06	0x00000020
VXEV07	0x00000040
VXEV08	0x00000080
VXEV09	0x00000100
VXEV10	0x00000200
VXEV11	0x00000400
VXEV12	0x00000800
VXEV13	0x00001000
VXEV14	0x00002000
VXEV15	0x00004000
VXEV16	0x00008000
VXEV17	0x00010000
VXEV18	0x00020000
VXEV19	0x00040000
VXEV20	0x00080000
VXEV21	0x00100000

Table 7-6 **VxWorks Event Flags** (cont'd)

Macro	Value
VXEV22	0x00200000
VXEV23	0x00400000
VXEV24	0x00800000

The functions that affect the contents of the events register are described in Table 7-7.

Table 7-7 **Functions That Modify the Task Events Register**

Function	Effect on the Task Events Register
eventReceive( )	Clears or leaves the contents of the task's events register intact, depending on the options selected.
eventClear( )	Clears the contents of the task's events register.
eventSend( )	Writes events to a tasks's events register.
semGive( )	Writes events to the tasks's events register, if the task is registered with the semaphore.
msgQSend( )	Writes events to a task's events register, if the task is registered with the message queue.

Receiving Events

A task can pend on one or more events, or simply check on which events have been received, with a call to **eventReceive( )**. Events are specified by using the value the relevant event flags for the function's **events** parameter (see *About Event Flags and the Task Events Register*, p. 138). The function provides options for waiting for one or all of those events, as well as options for how to manage unsolicited events, and for checking which events have been received prior to the full set being received.

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 (specified by the function's ***pEventsReceived** parameter), which 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.

Additional Steps for Receiving Events from Semaphores and Message Queues

In order for a task to receive events when a semaphore or a message queue is free (as opposed to simply another task), 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( )** function identifies the semaphore and the events that should be sent 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, events are sent to the registered task.

To request that event-sending be stopped, the registered task calls **semEvStop( )**.

Registration with a message queue is similar to registration with a semaphore. The **msgQEvStart( )** function identifies the message queue and the events that should be sent 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; and 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, events are sent to the registered task.

To request that event-sending be stopped, the registered task calls **msgQEvStop( )**.

Sending Events

Tasks and ISRs can send specific events to a task explicitly using **eventSend( )**, whether or not the receiving task is prepared to make use of them. Events are specified by using the value the relevant event flags for the function's **events** parameter (see *About Event Flags and the Task Events Register*, p.138).

Events are sent automatically to tasks that have registered for notification when semaphores and message queues are *free*—with **semEvStart( )** or **msgQEvStart( )**, respectively. 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( )**, events are not sent 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 an event is sent to a task to indicate that a semaphore or message queue 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 function 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.

For information about timeouts, see *7.19 About VxWorks API Timeout Parameters*, p. 149.

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 that events could not be sent to the registered task. Note that a failure to send a message or give a semaphore takes precedence over an events failure.

Inter-Process Communication With Events

For events to be used in inter-process communication, the objects involved in communication (tasks, semaphores, and message queues) must be public. For more information, see *7.18 Inter-Process Communication With Public Objects*, p.147.

Events Functions

The functions used for working with events are listed in Table 7-8.

Table 7-8 **Events Functions**

Function	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 functions, and code examples, see the API references for **eventLib**, **semEvLib**, and **msgQEvLib**.

Show Functions 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( )**

7.17 VxWorks Condition Variables

VxWorks condition variables provide a task-synchronization mechanism that combines thread-safe access to a resource with event signaling capabilities.

Thread-safe access to the resource is ensured by a standard VxWorks mutex semaphore that is released and re-acquired atomically during the operation of the associated condition variable. The typical use case for condition variables is the producer/consumer type of application, which allows:

- Thread-safe updating (under a mutex lock) of application data.
- Signaling for tasks waiting for a specific condition of the application data.
- Thread-safe checking for a condition (under a mutex lock) and wait (pend) until the condition is signaled.

VxWorks condition variables are functionally equivalent to POSIX condition variables (**pthread_condvar_wait( )**, **pthread_condvar_signal( )**, and so on). While POSIX condition variables only work in the context of POSIX threads, VxWorks condition variables work in the context of any VxWorks task, and they work in conjunction with native VxWorks mutex semaphores. For information about POSIX condition variables in VxWorks see *9.17 POSIX Pthread Mutexes and Condition Variables*, p.204.

Condition variable functions are not callable from interrupt service routines (ISRs).

VxWorks Configuration for Condition Variables

Note that no VSB configuration is required for condition variables.

Support for condition variables is provided with the **INCLUDE_CONDVAR** component. If you RTP support with the **INCLUDE_RTP** component, **INCLUDE_CONDVAR** is added automatically.

The **condVarShow( )** function is provided with the **INCLUDE_CONDVAR_SHOW** component.

Condition Variable APIs

Table 7-9**ConditionVariableFunctions**

Function	User Mode	Description
condVarCreate( )	yes	Create a condition variable.
condVarInitialize( )	no	Initialize a statically allocated condition variable. Kernel only.
condVarDelete( )	yes	Delete a condition variable; succeeds if there are no tasks waiting.
condVarDestroy( )	no	Delete a condition variable; flushes waiting tasks, if any.
condVarTerminate( )	no	Terminate a statically initialized condition variable.
condVarWait( )	yes	Pend on a condition variable until it is signaled.

Table 7-9**Condition Variable Functions** (cont'd) (cont'd)

Function	User Mode	Description
condVarSignal( )	yes	Signal a condition variable, releasing a pended task.
condVarBroadcast( )	yes	Release all tasks pended on a condition variable.
condVarShow( )	no	Show information about a condition variable.
condVarOpen( )	yes	Open a named condition variable.
condVarClose( )	yes	Close a named condition variable.
condVarUnlink( )	yes	Unlink a named condition variable.

For more information, see the **condVarlib**, **condVarOpen**, and **condVarShow** API reference entries.

System Viewer Events

The condition variables subsystem includes four System Viewer events.

Table 7-10**System Viewer Events for Condition Variables**

System Viewer Event	Emitted When
EVENT_CONDVAR_CREATE	Creation of a condition variable
EVENT_CONDVAR_WAIT	Waiting on a condition variable
EVENT_CONDVAR_SIGNAL	Signaling a condition variable
EVENT_CONDVAR_BROADCAST	Broadcast of a condition variable

Example Code

This example illustrates the use of a condition variables. The two functions, **varRead( )**() and **varWrite( )**() can be invoked by two different tasks, allowing them to synchronize when data is ready to read (signaled via one of the condition variables, **canRead**) and when new data can be written (signaled by way of the other condition variable, **canWrite**).

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

typedef enum state {READABLE, WRITABLE} STATE;

STATE varState = WRITEABLE; /* variable state */
int var;                    /* data to read or write */
CONDVAR_ID canRead;         /* signal READABLE state */
CONDVAR_ID canWrite;        /* signal WRITABLE satte */
SEM_ID     varMutex;        /* for mutual exclusion */


void appInitialize (void)
    {
    /* Create a mutual-exclusion semaphore. */

    varMutex = semMCreate (SEM_Q_PRIORITY);
    /* Create a condition variables */

    canRead = condVarCreate (CONDVAR_Q_PRIORITY);
    canWrite = condVarCreate (CONDVAR_Q_PRIORITY);
    }

STATUS varRead
    (
    int * pData
    )
    {
    semTake (varMutex, WAIT_FOREVER);

    /* If the condition is not true, wait on the condVar. */

    while (varState != READABLE)
        {
        condVarWait (canRead, varMutex, WAIT_FOREVER);
        }

    /* Now the variable is readable; the caller owns the mutex. */

    *pData = var;

    /* Reset the condition. */

    varState = WRITEABLE;

    semGive (varMutex);

    /* Signal the writer that new data can be written */

    condVarSignal (canWrite);

    return OK;
    }

STATUS varWrite
    (
    int data
    )
    {
    semTake (varMutex, WAIT_FOREVER);

    /* If the condition is not true, wait on the condVar. */

    while (varState != WRITEABLE)
        {
        condVarWait (canWrite, varMutex, WAIT_FOREVER);
        }

     /* now the variable is writable; the caller owns the mutex. */

    var = data;

    /* new data written, reset the variable state to readable */

    varState = READABLE;

    semGive (varMutex);

    /* Signal the reader that data as been written */

    condVarSignal (canRead);

    return OK;
    }
```

7.18 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 by defining them as public. Public objects are *visible* from all processes and the kernel, and can therefore be used for interprocess (and kernel-process) communication. 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 *6.7 Task Names and IDs*, p.92.

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 *6.7 Task Names and IDs*, p.92.

To create a named object, the appropriate *xyz***Open( )** API must be used, such as **semOpen( )**. When the function 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 created public kernel objects in the kernel, see the **msgQOpen**, **semOpen**, **taskOpen**, and **timerOpen** entries in the *VxWorks Kernel API Reference*. For 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*.

Example of Inter-process Communication With a Public Semaphore

The following code provides a simple example of using of a public semaphore to coordinate the activities of two RTPs. The broader context might be that the **take** RTP requires that some setup activity be performed by the **give** RTP before it runs. If the **take** RTP is started before the **give** RTP, it blocks until the **give** RTP runs and gives the semaphore; if it is started after the **give** RTP, it simply takes the semaphore and proceeds.

take.c RTP Application:

```
#include <vxWorks.h>
#include <stdio.h>
#include <semLib.h>
#include <sysLib.h>
#include <taskLib.h>

void main (void)
    {
    char *  semName = "/semIdTZsync";   /* Public semaphore */
    SEM_ID  semSyncRTP;

    /*
     * Create a binary semaphore to be used for synchronization
     * with the RTP.
     */

    semSyncRTP = semOpen (semName,  /* name of semaphore */
            SEM_TYPE_BINARY,/* type of semaphore */
            SEM_EMPTY,  /* initial state or initial count */
            SEM_Q_FIFO, /* semaphore options */
            OM_CREATE,  /* OM_CREATE, ... */
            NULL);      /* context value. Not used in VxWorks */


    printf("%s trying to take the semaphore %s\n", taskName(taskIdSelf()),
semName);
    semTake(semSyncRTP, WAIT_FOREVER);
    printf("\thave it\n");
    }
```

give.c RTP Application

```
#include <vxWorks.h>
#include <stdio.h>
#include <semLib.h>
#include <sysLib.h>
#include <taskLib.h>

void main (void)
    {
    char *  semName = "/semIdTZsync";   /* Public semaphore */
    SEM_ID  semSyncRTP;

    /*
     * Create a binary semaphore to be used for synchronization
     * with the RTP.
     */

    semSyncRTP = semOpen (semName,  /* name of semaphore */
            SEM_TYPE_BINARY,/* type of semaphore */
            SEM_EMPTY,  /* initial state or initial count */
            SEM_Q_FIFO, /* semaphore options */
            OM_CREATE,  /* OM_CREATE, ... */
            NULL);      /* context value. Not used in VxWorks */
    printf("%s giving the semaphore %s\n", taskName(taskIdSelf()), semName);
    semGive(semSyncRTP);
    printf("\tgave it\n");
    }
```

7.19 About VxWorks API Timeout Parameters

Various native VxWorks APIs take a timeout parameter whose value is specified in ticks—with **semTake( )**, **msgQReceive( )**, and so on. A timeout in ticks cannot equate exactly to a specific time in seconds.

For example, specifying a timeout of 1 means until the next system clock interrupt, which may occur at some point less than the full amount of time between interrupts. In order to ensure that the timeout is for at least the desired amount of time, add an additional tick to your timeout value. For example, if one tick is one millisecond, and you want a timeout of at least two milliseconds, use a timeout value of 3.

In contrast to native VxWorks APIs, the POSIX APIs provided with VxWorks use a timeout parameter whose value is specified in seconds—**sem_timedwait( )**, for example. Internally, when the time value is converted to ticks, the tick count is automatically increased by one to ensure that the timeout is at least for the specified amount of time.

You can use **sysClockRateGet( )** to determine the number of ticks per second. You can change the clock rate with the **SYS_CLK_RATE** configuration parameter.

7.20 About Object Ownership and Resource Reclamation

All objects (such as semaphores) 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 functions). 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.

In the kernel, the **objShowAll( )** show function can be used to display information about ownership relations between objects. In RTPs, the **objHandleShow( )** show function can be used to display information about ownership relations between objects in a process.

8
Signals, ISRs, and Watchdog Timers

8.1 About 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 functions that allow a task to respond to a specific signal in whatever way is useful for your application. For information about signal handlers, see *8.6 Signal Handlers*, p. 161.

A kernel task or interrupt service routine (ISR) can raise a signal for a specific task or process. In the kernel, signal generation and delivery runs in the context of the task or ISR that generates the signal. In accordance with the POSIX standard, a signal sent to a process is handled by the first available task that has been set up to handle the signal in the process.

Each kernel task has a signal mask associated with it. The signal mask determines which signals the task accepts. By default, the signal mask is initialized with all signals unblocked (there is no inheritance of mask settings in the kernel). The mask can be changed with **sigprocmask( )**.

Signal handlers in the kernel can be registered for a specific task. A signal handler executes in the receiving task's context and makes use of that task's execution stack. The signal handler is invoked even if the task is blocked (suspended or pended).

VxWorks provides a software signal facility that includes POSIX functions, UNIX BSD-compatible functions, and native VxWorks functions. The POSIX-compliant signal interfaces include both the basic signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b.

Additional, non-POSIX APIs provide support for signals between kernel and user applications. These non-POSIX APIs are: **taskSigqueue( )**, **rtpSigqueue( )**, **rtpTaskSigqueue( )**, **taskKill( )**, **rtpKill( )**, **rtpTaskKill( ),** and **taskRaise( )**.

In the VxWorks kernel—for backward compatibility with prior versions of VxWorks—the POSIX-like API that would take a process identifier as one of their parameters, take a task identifier instead.

NOTE: The **SIGEV_THREAD** option is not supported in the kernel.

NOTE: Wind River recommends that you do not use both POSIX APIs and VxWorks APIs in the same application. Doing so may make a POSIX application non-conformant.

An RTP task (user-mode) can raise a signal for any of the following:

- itself
- any other task in its process
- any public task in any process 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 *Public Naming of Tasks for Inter-Process Communication*, p.93.) 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 functions 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 functions and native VxWorks functions. 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**.

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 *7.16 VxWorks Events*, p.138.

8.2 VxWorks Configuration for Signals

By default, VxWorks includes the basic signal facility component **INCLUDE_SIGNALS**. Other components provide additional facilities.

Configuration for Signal Events

To include the signal event facility, configure VxWorks with the **INCLUDE_SIGEVENT** component. For information about signal events, see *8.5 Signal Events*, p.160.

Configuration for Kernel POSIX Queued Signals

To include kernel POSIX queued signals in the system, configure VxWorks with the **INCLUDE_POSIX_SIGNALS** component. This component automatically initializes POSIX queued signals with **sigqueueInit( )**. The **sigqueueInit( )** function allocates buffers for use by **sigqueue( )**, which requires a buffer for each currently queued signal. A call to **sigqueue( )** fails if no buffer is available.

The maximum number of queued signals in the kernel is set with the configuration parameter **NUM_SIGNAL_QUEUES**. The default value is 16.

Configuration for RTPs

Note that the **SIGEV_THREAD** option is only supported in real-time processes (and not in the kernel). The **SIGEV_THREAD** option requires that—in addition to the **INCLUDE_SIGEVENT** component —VxWorks 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.

8.3 Basic Signal Functions

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( )** function 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( )** function, which sends a signal to a task.

Kernel Signal Functions

Table 8-1 **POSIX and BSD Kernel Function**

POSIX 1003.1b Compatible Function	UNIX BSD Compatible Function	Description
signal( )	**signal( )**	Specifies the handler associated with a signal.
raise( )	N/A	Sends a signal to yourself.
sigaction( )	**sigvec( )**	Examines or sets the signal handler for a signal.
sigsuspend( )	**pause( )**	Suspends a task until a signal is delivered.
sigpending( )	N/A	Retrieves a set of pending signals blocked from delivery.

sigemptyset( )
sigfillset( )
sigaddset( )
sigdelset( )
sigismember( )
	N/A	Manipulates a signal mask.

sigprocmask( )	N/A	Sets the mask of blocked signals.
sigprocmask( )	N/A	Adds to a set of blocked signals.

VxWorks also provides additional functions that serve as aliases for POSIX functions, such as **rtpKill( )**, that provide for sending signals from the kernel to processes.

For more information about signal functions, see the API reference for **sigLib** and **rtpSigLib**.

User Mode Signal Functions

Table 8-2 **Basic POSIX 1003.1b Signal Functions**

Function	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 functions, see the API reference for **sigLib**.

8.4 Queued Signal Functions

The **sigqueue( )** family of functions provides an alternative to the **kill( )** family of functions for sending signals.

The important differences between the two are the following:

- The **sigqueue( )** function 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( )** function enables the queueing of multiple signals for any task. The **kill( )** function, 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 is 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 function **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 function **sigtimedwait( )** is similar, except that it can time out.

The basic queued signal functions are described in Table 8-3. For detailed information on signals, see the API reference for **sigLib**.

Table 8-3 **POSIX 1003.1b Queued Signal Functions**

Function	Description
sigwaitinfo( )	Waits for a signal.
sigtimedwait( )	Waits for a signal with a timeout.

Additional non-POSIX VxWorks queued signal functions are described in Table 8-4. These functions are provided for assisting in porting VxWorks 5.*x* kernel applications to processes. The POSIX functions described in Table 8-3 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 functions—**taskKill( )**, **rtpKill( )**, and **rtpTaskKill( )**.

Table 8-4 **Non-POSIX Queued Signal Functions**

Function	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 8-1 **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 function 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 function 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 function to send a queued signal
```

```
*
* This function 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( )** function provided in this code is a helper function used to send a queued signal.

Kernel Application Use

To use the code as a kernel application, VxWorks must be configured with **BUNDLE_NET_SHELL** and **BUNDLE_POSIX**.

To send a queued signal to a kernel task:

1. Build the code with the VxWorks image or as a downloadable kernel module. Boot the system. If the code is not linked to the system image, load the module from the kernel shell.
2. Spawn a task with **main( )** as the entry point.

3. Send a queued signal to the spawned kernel task, using **sig( )**; the first parameter is the task ID and the second is the signal value.

From the kernel shell, for example:

```
-> ld < signal_ex.o
value = 8382064 = 0x7fe670
-> sp main
Task spawned: id = 0x7fd620, name = t1
value = 8377888 = 0x7fd620
-> Task 0x7fd620 installed signal handler for signal # 30.  Ready for signal.
-> sig 0x7fd620, 20
value = 0 = 0x0
-> Task 0x7fd620 got signal # 30  signal value 20
sig 0x7fd620, 20
value = 0 = 0x0
-> Task 0x7fd620 got signal # 30  signal value 20
```

RTP Application Use

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.
3. Determine the RTP ID using **rtpShow( )**.
4. From a kernel task (note that the kernel shell runs as a kernel task), use the **sig( )** function to send a queued signal to an RTP, where the first parameter is the RTP ID and the second is the signal value.

From the kernel shell, for example:

```
-> rtpSp "signal_ex.vxe"
value = 10531472 = 0xa0b290
-> Task 0x10000 installed signal handler for signal # 30.  Ready for signal.
-> rtpShow

       NAME              ID          STATE      ENTRY ADDR  OPTIONS   TASK CNT
-------------------- ---------- --------------- ---------- ---------- --------
signal_ex.vxe          0xa0b290 STATE_NORMAL    0x10000298        0x1        1

value = 1 = 0x1
-> sig 0xa0b290, 50
value = 0 = 0x0
-> Task 0x10000 got signal # 30  signal value 50
```

8.5 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 functions 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 **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 user space (RTPs), 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).

8.6 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 function 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 functions listed in Table 8-5. 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( )** functions 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 implemented.
- 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 8-5 **Functions Callable by Signal Handlers**

Library	Functions
bLib	All functions
errnoLib	**errnoGet( )**, **errnoSet( )**
eventLib	**eventSend( )**
logLib	**logMsg( )**
lstLib	All functions except **lstFree( )**
msgQLib	**msgQSend( )**
rngLib	All functions 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( )**
vxAtomicLib	All functions.
vxCpuLib	**vxCpuIndexGet( )**, **vxCpuIdGet( )**, **vxCpuPhysIndexGet( )**, **vxCpuIdToPhysIndex( )**, **vxCpuPhysIndexToId( )**, **vxCpuReservedGet( )**, **CPU_LOGICAL_TO_PHYS( )**, and **CPU_PHYS_TO_LOGICAL( )**
ansiString	All functions.

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.

- For RTP applications, modify the parent process with an appropriate error return value.
- For kernel applications, notify the kernel with an appropriate error return value.

Mutual exclusion between signal handlers and tasks must be managed with care. In general, users should avoid the following activity in signal handlers:

- Taking mutual exclusion (such as semaphores) resources that can also be taken by any other element of the application code. This can lead to deadlock.
- Modifying any shared data memory that may have been in the process of modification by any other element of the application code when the signal was delivered. This compromises mutual exclusion and leads to data corruption.
- Using **longjmp( )** to change the flow of task execution. If **longjmp( )** is used in a signal handler to re-initialize a running task, you must ensure that the signal is not sent to the task while the task is holding a critical resource (such as a kernel mutex). For example, if a signal is sent to a task that is executing **malloc( )**, the signal handler that calls **longjmp( )** could leave the kernel in an inconsistent state.

These scenarios are very difficult to debug, and should be avoided. One safe way to synchronize other elements of the application code and a signal handler is to set up dedicated flags and data structures that are set from signal handlers and read from the other elements. This ensures a consistency in usage of the data structure. In addition, the other elements of the application code must check for the occurrence of signals at any time by periodically checking to see if the synchronizing data structure or flag has been modified in the background by a signal handler, and then acting accordingly. The use of the **volatile** keyword is useful for memory locations that are accessed from both a signal handler and other elements of the application.

Taking a mutex semaphore in a signal handler is an especially bad idea. Mutex semaphores can be taken recursively. A signal handler can therefore easily re-acquire a mutex that was taken by any other element of the application. Since the signal handler is an asynchronously executing entity, it has thereby broken the mutual exclusion that the mutex was supposed to provide.

Taking a binary semaphore in a signal handler is an equally bad idea. If any other element has already taken it, the signal handler will cause the task to block on itself. This is a deadlock from which no recovery is possible. Counting semaphores, if available, suffer from the same issue as mutexes, and if unavailable, are equivalent to the binary semaphore situation that causes an unrecoverable deadlock.

On a general note, the signal facility should be used only for notifying/handling exceptional or error conditions. Usage of signals as a general purpose IPC mechanism or in the data flow path of an application can cause some of the pitfalls described above.

8.7 About Interrupt Service Routines: ISRs

Hardware interrupt handling is of key significance in real-time systems, because it is usually through interrupts that the system is informed of external events. Interrupt service routines (ISRs)—which are also known as interrupt handlers—can be implemented in the kernel to provide the appropriate response to interrupts.

You can attach ISRs to any system hardware interrupts that are not used by VxWorks. VxWorks provides a function that allows for creating custom ISRs by connecting C functions to interrupts. When an interrupt occurs, the connected C function is called at interrupt level. When the interrupt handling is finished, the connected function returns.

In order to achieve the fastest possible response to interrupts, VxWorks interrupt service routines (ISRs) in run in a special context of their own, outside any task's context. VxWorks runs ISRs when the associated interrupt occurs; there is no deferral of ISR execution unless the system has been specifically configured to do so.

8.8 VxWorks Configuration for ISRs

Support for ISRs is provided by default. However, the interrupt stack can be configured with regard to size and additional features. In addition, ISR deferral support and show function support can be added to the system.

Configuring the Interrupt Stack

All ISRs use the same *interrupt stack*. This stack is allocated and initialized by the system at startup according to specified configuration parameters. It must be large enough to handle the worst possible combination of nested interrupts.

The size of the interrupt stack is defined with the **ISR_STACK_SIZE** parameter. For information about determining the size of the stack, see *Interrupt Stack Size and Overruns*, p. 169.

CAUTION: Some architectures do not permit using a separate interrupt stack, and ISRs use the stack of the interrupted task. With such architectures, make sure to create tasks with enough stack space to handle the worst possible combination of nested interrupts *and* the worst possible combination of ordinary nested calls. See the VxWorks reference for your BSP to determine whether your architecture supports a separate interrupt stack. If it does not, also see *Task Stack Protection*, p. 96.

Interrupt Stack Filling

By default, interrupt (and task) stacks are filled with 0xEE. Filling stacks is useful during development for debugging with the **checkStack( )** function.

For deployed systems, it is often preferable not to fill the interrupt stack because not filling provides better performance. Use use the **VX_GLOBAL_NO_STACK_FILL** configuration parameter to disable stack filling for all interrupts (and tasks) in the system.

Interrupt Stack Protection

Systems can be configured with the **INCLUDE_PROTECT_INTERRUPT_STACK** component to provide guard zone protection at the start and end of the interrupt stack, as long as the MMU is enabled.

An overrun guard zone indicates when a task goes beyond the end of its predefined stack size, which can be used to ensure that the integrity of the system is not compromised by corruption of other stacks or data. For architectures that grow down, an under-run guard zone typically prevents buffer overflows from corrupting memory above the stack. The CPU generates an exception when a task attempts to access any of the guard zones. The size of a stack is always rounded up to a multiple the MMU page size when a guard zone is inserted.

The sizes of the guard zones are defined by the following configuration parameters:

- **INTERRUPT_STACK_OVERFLOW_SIZE** for interrupt stack overflow size.
- **INTERRUPT_STACK_UNDERFLOW_SIZE** for interrupt stack underflow size.

The size of a stack is always rounded up to a multiple of the MMU page size when a guard zone is added. If the size of the system becomes an issue, the component can be removed for final testing and the deployed system. Setting a parameter to zero prevents insertion of the corresponding guard zone.

Adding Show Function Support

To add support for the **isrShow( )** function, add the **INCLUDE_ISR_SHOW** component. See *8.13 ISR Information at Runtime*, p.172.

8.9 Facilities Available for ISRs

All VxWorks utility libraries, such as the linked-list and ring-buffer libraries, can be used by ISRs. There are however, restrictions on the functions that can be called from ISRs.

The global variable **errno** is saved and restored as a part of the interrupt enter and exit code generated by the **intConnect( )** facility. Thus, **errno** can be referenced and modified by ISRs as in any other code.

Table 8-6 lists functions that can be called from ISRs.

Table 8-6 **Functions Callable by ISRs**

Library	Function
bLib	All functions
errnoLib	**errnoGet( )**, **errnoSet( )**
eventLib	**eventSend( )**
fppArchLib	**fppSave( )**, **fppRestore( )**

Table 8-6 **Functions Callable by ISRs** (cont'd)

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

8.10 ISR Coding and Debugging

The most basic guideline for ISRs is that they should be as short possible. Time-consuming activities should not be undertaken with ISRs. There are also numerous restrictions: ISRs must not call functions that might cause the caller to block, or call functions that use a floating-point coprocessor; they have limitations with regard to C++ use; and so on.

No Blocking Functions

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

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

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

ISRs also must not perform I/O through VxWorks drivers. Although there are no inherent restrictions in the I/O system, most device drivers require a task context because they might block the caller to wait for the device. An important exception is the VxWorks pipe driver, which is designed to permit writes by ISRs. VxWorks also provides several functions that can be called from an ISR to print messages to the system console: **logMsg( )**, **kprintf( )**, and **kputs( )**. For more information about using these functions in ISRs, see *8.10 ISR Coding and Debugging*, p.166.

No Floating-Point Coprocessor Functions

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

C++ Code Limitations

Be particularly careful when using C++ for an ISR or when invoking a C++ method from an ISR written in C or assembly. Some of the issues involved in using C++ include the following:

- The VxWorks **intConnect( )** function require the address of the function to execute when the interrupt occurs, but the address of a non-static member function cannot be used, so static member functions must be implemented.
- Objects cannot be instantiated or deleted in ISR code.
- C++ code used to execute in an ISR should restrict itself to Embedded C++. No exceptions nor run-time type identification (RTTI) should be used.

No Direct Shared Data Region Access

An ISR should not be designed to access a shared data region directly. An ISR inherits the memory context of the task that it has preempted, and if that task happens to be in a memory context (kernel or process) that does not include the shared data region in question, it cannot access that memory, and an exception will result.

In order to access a shared data region reliably, an ISR must make use of a task that has already been created (ISRs cannot create tasks themselves) in a memory context that includes the shared data region (for example, from a process that is attached to the region). The task can then perform the required operation on the region after the ISR has terminated.

Interrupt-to-Task Communication

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

Shared Memory and Ring Buffers
: ISRs can share variables, buffers, and ring buffers with task-level code.

Semaphores
: ISRs can give semaphores (except for mutual-exclusion semaphores and VxMP shared semaphores) that tasks can take and wait for.

Message Queues
: ISRs can send messages to message queues for tasks to receive (except for shared message queues using VxMP). If the queue is full, the message is discarded.

Pipes
: ISRs can write messages to pipes that tasks can read. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message written is discarded because the ISR cannot block. ISRs must not invoke any I/O function on pipes other than **write( )**.

Signals
: ISRs can *signal* tasks, causing asynchronous scheduling of their signal handlers.

VxWorks Events
: ISRs can send VxWorks events to tasks.

Reserving High Interrupt Levels

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

⚠ **CAUTION:** Some hardware prevents masking certain interrupt levels; check the hardware manufacturer's documentation.

Restrictions for ISRs at High Interrupt Levels

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

- The ISR can be connected only with **intVecSet( )**.
- The ISR cannot use any VxWorks operating system facilities that depend on interrupt locks for correct operation. The effective result is that the ISR cannot safely make any call to any VxWorks function, except reboot.

For more information, see the *VxWorks Architecture Supplement* for the architecture in question.

WARNING: The use of NMI with any VxWorks functionality, other than reboot, is not recommended. Functions marked as *interrupt safe* do not imply they are NMI safe and, in fact, are usually the very ones that NMI functions must not call (because they typically use **intLock( )** to achieve the interrupt safe condition).

Restrictions on I/O Use

In general, ISRs must not perform I/O through VxWorks drivers because they might cause the caller to block (for more information, see *The most basic guideline for ISRs is that they should be as short possible. Time-consuming activities should not be undertaken with ISRs. There are also numerous restrictions: ISRs must not call functions that might cause the caller to block, or call functions that use a floating-point coprocessor; they have limitations with regard to C++ use; and so on.*, p.166). This means that standard I/O functions such as **printf( )** and **puts( )** cannot be used to debug ISRs. The basic programmatic methods available for debugging ISRs are as follows:

- Coding an ISR to use global variables that are updated with relevant data. The shell (kernel or host) can then be used to display the values of the variables at runtime.
- Using one of the functions that can be called from an ISR to write a message (including variable states) to the console. The **logMsg( )**, **kprintf( )**, and **kputs( )** functions can all be called from an ISR.

The advantages of using global variables (and the shell to display their values) are simplicity and minimal performance impact on the execution of the ISR. The disadvantage is that if the target hangs due a bug in the ISR, the shell cannot be used to display variable values.

The advantages of using **logMsg( )** are that it can be used to print messages to the console automatically, and it has less of an effect on the performance of the ISR than either **kprintf( )** or **kputs( )**. The disadvantage is that if the target hangs shortly after the function is called, the message string may not be displayed on the console. This is because of the asynchronous operation in which **logMsg( )** first writes to a message queue and then the logging task packages the message and sends it to the console.

The advantage of the **kprintf( )** and **kputs( )** functions is that they output messages synchronously (using polled mode), and can therefore indicate precisely how far an ISR has progressed in its execution when a bug manifests itself. (They can also be used during the kernel boot sequence and from task switch hooks.) The disadvantage of using these functions is that they have a notable effect on the performance of the ISR because they output message strings over the serial port.

Also note that you can use global variables to affect the flow control of ISRs (quite apart from the method used to output debug information).

Interrupt Stack Size and Overruns

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

In addition to experimenting with stack size, you can also configure and test systems with guard zone protection for interrupt stacks (for more information, see *Interrupt Stack Protection*, p.165).

Exceptions at Interrupt Level

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

The VxWorks boot loader tests for the presence of the exception description in low memory and if it is detected, display it on the system console. The boot loader's **e** command re-displays the exception description.

One example of such an exception is the following message:

```
workQPanic: Kernel work queued from ISR overflowed.
```

For more information, see *ISRs and Kernel Work Queue Panic*, p.170.

ISRs and Kernel Work Queue Panic

The error message associated with a kernel work queue panic (also known as a work queue overflow) is as follows:

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

In order to help reduce the occurrences of work queue overflows, system architects can use the **WIND_JOBS_MAX** kernel configuration parameter to increase the size of the kernel work queue. However in most cases this is simply hiding the root cause of the overflow. For more information see *8.14 ISRs and Work Queue Panic*, p.172.

8.11 System Clock ISR Modification

You can add application-specific processing to **usrClock( )**.

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

If you add application-specific processing to **usrClock( )**. However, you should keep in mind that this is executed in interrupt context, so only limited functions can be safely called. See *The most basic guideline for ISRs is that they should be as short possible. Time-consuming activities should not be undertaken with ISRs. There are also numerous restrictions: ISRs must not call functions that might cause the caller to block, or call functions that use a floating-point coprocessor; they have limitations with regard to C++ use; and so on.*, p.166 for a list of functions that can be safely used in interrupt context.

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

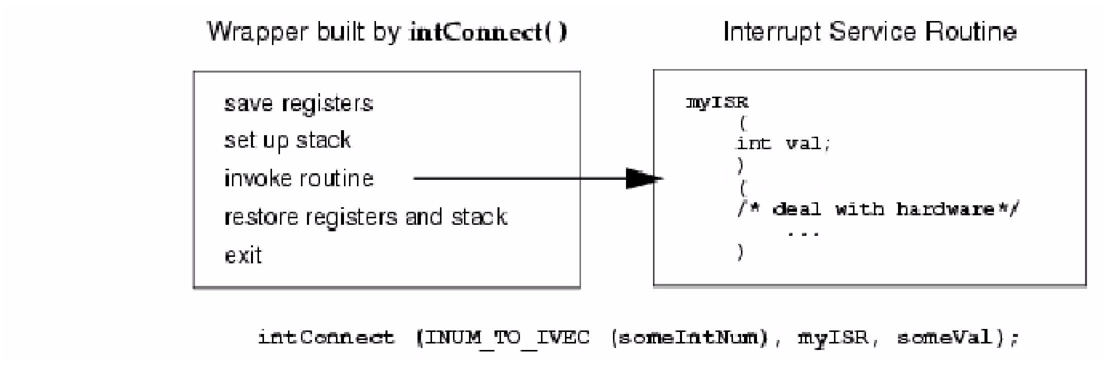
8.12 ISR Connection to Interrupts

VxWorks provides the function **intConnect( )**, which allows C functions to be connected to any interrupt.You can use system hardware interrupts other than those used by VxWorks.

The arguments to **intConnect( )** are the byte offset of the interrupt vector to connect to, the address of the C function to be connected, and an argument to pass to the function. When an interrupt occurs with a vector established in this way, the connected C function is called at interrupt level with the specified argument. When the interrupt handling is finished, the connected function returns. A function connected to an interrupt in this way is called an *interrupt service routine* (ISR).

Interrupts cannot actually vector directly to C functions. Instead, **intConnect( )** builds a small amount of code that saves the necessary registers, sets up a stack entry (either on a special interrupt stack, or on the current task's stack) with the argument to be passed, and calls the connected function. On return from the function it restores the registers and stack, and exits the interrupt; see Figure 8-1.

Figure 8-1 **Function Built by intConnect( )**


Wrapper built by intConnect( )
Interrupt Service Routine
save registers
set up stack
invoke routine
restore registers and stack
exit

```
myISR
    (
    int val;
    )
    (
    /* deal with hardware*/
       ...
    )
```

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

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

Table 8-7 lists the interrupt functions provided in **intLib** and **intArchLib**.

Table 8-7 **Interrupt Functions**

Function	Description
intConnect( )	Connects a C function to an interrupt vector.
intContext( )	Returns **TRUE** if called from interrupt level.
intCount( )	Gets the current interrupt nesting depth.
intLevelSet( )	Sets the processor interrupt mask level.
intLock( )	Disables interrupts.
intUnlock( )	Re-enables interrupts.
intVecBaseSet( )	Sets the vector base address.
intVecBaseGet( )	Gets the vector base address.

Table 8-7 **Interrupt Functions** (cont'd)

Function	Description
intVecSet( )	Sets an exception vector.
intVecGet( )	Gets an exception vector.

For information about interrupt locks and latency, see *7.3 Interrupt Locks*, p. 113.

NOTE: The **intLock( )** and **intUnlock( )** functions are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including the ISR-callable spinlock, which default to **intLock( )** and **intUnlock( )** behavior in a UP system. For more information, see *ISR-Callable Spinlocks*, p. 423 and *18.18 Code Migration for VxWorks SMP*, p. 449.

8.13 ISR Information at Runtime

A kernel object is created for each ISR that is linked with VxWorks. ISR objects provide a means for managing information about all the ISRs in a system.

See the **isrLib** and **isrShow** API references for information about the functions used for getting information about ISRs.

The **INCLUDE_ISR_SHOW** component provides **isrShow**, and the **INCLUDE_ISR_OBJECT** provides **isrLib**.

8.14 ISRs and Work Queue Panic

If ISRs queue up kernel work at a very high rate before the kernel can process them, the queue can fill up, and failed attempt to add a new work item to the queue is a fatal error called *work queue panic*.

To achieve fast interrupt response time, the VxWorks kernel does not lock interrupts while processing critical sections, such as task state changes, context switches, rescheduling, and so on. With interrupts enabled, ISRs can execute as soon as possible. If an interrupt occurs while the kernel is executing a critical section (in the case of VxWorks SMP this could be on same CPU or another CPU), and the ISR executes a kernel function that also has to execute a critical section (such as the **semGive( )**, **msgQSend( )**, or **eventSend( )** function), then the kernel delays critical section execution to a later time when it is safe to execute. This is accomplished using the kernel *work queue*. The queue effectively serializes kernel-critical sections without blocking the execution of non-critical sections of the ISR handler.

For performance reasons, the work queue is a circular buffer of fixed size. You can configure the size with the **WIND_JOBS_MAX** configuration parameter; the value must be a power of two.

Work Queue Panic

If ISRs queue up kernel work at a very high rate (in bursts) before the kernel can process them, the queue can fill up. After the queue is full, a failed attempt to add a new work item to the queue is considered a fatal error, called *work queue panic*. VxWorks reboots and logs an error message through the system exception message area and the error detection and reporting facility (*13. Error Detection and Reporting*).

Generally, a work queue panic is the result of improperly written device drivers, such as:

- *Failing to acknowledge an interrupt*. If an interrupt handler fails to acknowledge an interrupt, the result is continuous interrupt generation.
- *Doing too much in interrupt context*. For the best real-time behavior, you should defer complex operations to a properly prioritized task that you associate to the interrupt source. The ISR should wake up the task using a single kernel call (for example, the **semGive( )** function).
- *Handling device event bursts as individual interrupts*. You should handle bursts in a loop in task context after disabling the device-specific interrupt. When you no longer detect un-handled events and the task is about to transition to pended, you can enable the device interrupt source again.

8.15 About Watchdog Timers

VxWorks provides a watchdog-timer mechanism in the kernel that allows any C function to be connected to a timer with a specified time delay.

Watchdog timers are maintained as part of the system clock ISR. Functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. Restrictions on ISRs also apply to functions connected to watchdog timers.

(For information about POSIX timers, see *9.11 POSIX Clocks and Timers*, p. 190.)

The functions in Table 8-8 are provided by the **wdLib** library.

Table 8-8 **Watchdog Timer Calls**

Function	Description
wdCreate( )	Allocates and initializes a watchdog timer.
wdDelete( )	Terminates and deallocates a watchdog timer.
wdStart( )	Starts a watchdog timer.
wdCancel( )	Cancels a currently counting watchdog timer.

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

Example 8-2 **Watchdog Timers**

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

/* includes */
#include <vxWorks.h>
#include <logLib.h>
#include <wdLib.h>

/* defines */
#define  SECONDS (3)

WDOG_ID myWatchDogId;
task (void)
    {
    /* Create watchdog */
    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);

    /* Set timer to go off in SECONDS - printing a message to stdout */
    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                 "Watchdog timer just expired\n") == ERROR)
        return (ERROR);
    /* ... */
    }
```

For information about POSIX timers, see *9.11 POSIX Clocks and Timers*, p.190.

Static Instantiation of Watchdog Timers

The **wdCreate( )** function performs a dynamic, two-step operation, in which memory is allocated for the watchdog timer object at runtime, and then the object is initialized. Watchdog timers (and other VxWorks objects) can also be statically instantiated—which means that their memory is allocated for the object at compile time—and the object is then initialized at runtime.

The macro **VX_WDOG** declares a watchdog timer at compile time. It takes one parameter, the name of the watchdog. The function **wdInitialize( )** is used at runtime to initialize the watchdog and make it ready for use. The same watchdog-timer name must be used with the macro and function. For example:

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

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

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

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

For general information about static instantiation, see *1.5 Static Instantiation of Kernel Objects*, p.7.

Inter-Process Communication With Public Watchdog Timers

VxWorks watchdog 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 *7.18 Inter-Process Communication With Public Objects*, p.147 and the **timerOpen( )** entry in the *VxWorks Kernel API Reference*.

9
POSIX Facilities

9.1 About VxWorks POSIX Support

VxWorks provides extensive POSIX support for both kernel and user mode (RTP) applications.

Note that the only VxWorks file system that is POSIX-compliant is the Highly Reliable File System (for more information, see the *VxWorks 7 File Systems Programmer's Guide*.

For detailed information about POSIX standards and facilities, see The Open Group Web sites at **http://www.opengroup.org** and **http://www.unix.org**.

Kernel POSIX Support

VxWorks provides extensive POSIX support in many of its native kernel libraries. To facilitate application portability, VxWorks provides additional POSIX interfaces as optional components. In the kernel, VxWorks implements some of the traditional interfaces described by the POSIX standard IEEE Std 1003.1 (POSIX.1) as well as many of the real-time interfaces in the POSIX.1 optional functionality.

RTP POSIX Support

The VxWorks user space (RTP) 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.

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 functions to facilitate porting code to VxWorks. The functions otherwise provide no useful function—pages are always locked in VxWorks systems (for more information see *9.14 POSIX Page-Locking Interface*, p.195).

- VxWorks tasks are scheduled on a system-wide basis; processes themselves cannot be scheduled. As a consequence, while POSIX access functions allow two values for contention scope (**PTHREAD_SCOPE_SYSTEM** and **PTHREAD_SCOPE_PROCESS**), only system-wide scope is implemented in VxWorks for these functions (for more information, see *9.15 POSIX Threads: Pthreads*, p.196 and *9.18 POSIX and VxWorks Scheduling*, p.208).

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

With some exceptions, 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 *Comparison of POSIX and VxWorks Semaphores*, p.218, although POSIX semaphores are also implemented in VxWorks. VxWorks extensions to POSIX are identified as such.

VxWorks POSIX signal functions, as well as BSD-compatible functions and native VxWorks functions in the kernel, are discussed in *8.1 About Signals*, p.152 and related topics.

POSIX memory management facilities are discussed in *10. Memory Management*.

9.2 VxWorks Configuration With POSIX Facilities

VxWorks provides extensive POSIX support for the kernel with many of its libraries, but the default configuration of VxWorks does not include many other POSIX facilities that may be used. Optional VxWorks components provide support for individual POSIX libraries.

VSB Options for POSIX Facilities

Source code support for POSIX facilities is provided with the **CORE_IO** VSB option, and the **SYSCALLS** option for RTP-specific features. Both of these options are selected by default.

Basic POSIX VIP Components

General POSIX support for the kernel can be provided with the **BUNDLE_POSIX** component bundle. If memory constraints require a finer-grained configuration, individual components can be used for selected features. See the configuration instructions for individual POSIX features throughout this chapter for more information in this regard.

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**.

9.3 POSIX PSE52 Support

The components required for POSIX PSE52 support within processes (RTPs) 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 the *VxWorks 7 File Systems Programmer's Guide*.
- The appropriate device driver component —for example **INCLUDE_XBD_RAMDRV**—and **INCLUDE_XBD_BLK_DEV** if the device driver requires it (that is, if it is not XBD-compatible).

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 the discussion of HRFS and POSIX PSE53 in the *VxWorks 7 File Systems Programmer's Guide*.

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 the *VxWorks 7 File Systems Programmer's Guide*. For information about null devices, see *12.7 Null Devices*, p.328.

CAUTION: Do not include the **vxWorks.h** header file in your RTP application if it must conform to the POSIX PSE52 profile. The **vxWorks.h** header file is required for non-POSIX VxWorks facilities, but it also makes applications non-conformant with the PSE52 profile.

VxWorks PSE52 Support and POSIX Functionality

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 *9.7 POSIX Header Files*, p. 185.

⚠ **CAUTION:** If a PSE52-conforming application relies on C99 constructs, the Wind River Diab Compiler must be used. The version of the GNU compiler provided with this release does not support the C99 language constructs.

9.4 VxWorks Components for POSIX Facilities

VxWorks provides components that must be configured in the kernel to provide support for specific POSIX facilities.

Table 9-1 **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**
Pipes	**INCLUDE_POSIX_PIPES**	**INCLUDE_POSIX_PIPES**
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**

Table 9-1 **VxWorks Components Providing POSIX Facilities** (cont'd)

POSIX Facility	**Required VxWorks Component**	
	for Kernel	**for Processes**
Signals	**INCLUDE_POSIX_SIGNALS**	N/A
System logging	**INCLUDE_SYSLOG**	**INCLUDE_SYSLOG**
Timers	**INCLUDE_POSIX_TIMERS**	**INCLUDE_POSIX_TIMERS**
Trace	N/A	INCLUDE_POSIX_TRACE
POSIX PSE52 support	N/A	BUNDLE_RTP_POSIX_PSE52

9.5 POSIX Libraries

Many POSIX-compliant libraries are provided for VxWorks.

Table 9-2 **POSIX Libraries**

Functionality	**Library**
Asynchronous I/O	**aioPxLib**
Buffer manipulation	**bLib**
Clock facility	**clockLib**
Directory handling	**dirLib**
Environment handling	C Library
Environment information	**sysconf( ) confstr( )**, and **uname( )**. User-level only.
File duplication	**ioLib**. User-level only.
File duplication	**iosLib**. Kernel-only.
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

Table 9-2 **POSIX Libraries** (cont'd)

Functionality	**Library**
Math	C Library
Memory allocation	**memLib**. User-level only.
Memory allocation	**memLib** and **memPartLib**. Kernel-only.
Network/Socket APIs	network libraries
String manipulation	C Library
System logging	**syslogLib**
Trace facility	**pxTraceLib**
Wide character support	C library. User-level only.

User-Level Standard C Library: libc

The standard C library for user mode (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*.

⚠ **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** functions to manipulate the system's tick counter if you are also using **clockLib** functions, do not use the **taskLib** API to change the priority of a POSIX thread instead of the **pthread** API, and so on.

9.6 POSIX Support Check at Run-time From an RTP

A POSIX user-mode application can use the **sysconf( )** and the **confstr( )** function at run-time to determine the status of POSIX support in the system.

The **sysconf( )** function 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( )** function returns a string associated with a system variable. With this release, the **confstr( )** function returns a string only for the system's default path.

In addition **uname( )** function 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.

9.7 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.

POSIX Header Files for Kernel Applications

The POSIX header files available for the kernel development environment are listed in Table 9-3.

Table 9-3 **POSIX Header Files**

Header File	Description
aio.h	asynchronous input and output
assert.h	verify program assertion
ctype.h	character types
dirent.h	format of directory entries
errno.h	system error numbers
fcntl.h	file control options
limits.h	implementation-defined constants
locale.h	category macros
math.h	mathematical declarations
mqueue.h	message queues
pthread.h	pthreads
sched.h	execution scheduling
semaphore.h	semaphores
setjmp.h	stack environment declarations
signal.h	signals
stdio.h	standard buffered input/output
stdlib.h	standard library definitions
string.h	string operations
sys/mman.h	memory management declarations
sys/resource.h	definitions for XSI resource operations
sys/stat.h	data returned by the **stat( )** function
sys/types.h	data types
sys/un.h	definitions for UNIX domain sockets

Table 9-3 **POSIX Header Files** (cont'd)

Header File	Description
time.h	time types
trace.h	trace facility
unistd.h	standard symbolic constants and types
utime.h	access and modification times structure

POSIX Header Files for RTP Applications

The POSIX header files available for the user development environment are listed in Table 9-4. The *PSE52* column indicates compliance with the POSIX PSE52 profile.

CAUTION: Do not include the **vxWorks.h** header file in your RTP application if it must conform to the POSIX PSE52 profile. The **vxWorks.h** header file is required for non-POSIX VxWorks facilities, but it also makes applications non-conformant with the PSE52 profile.

Table 9-4 **POSIX Header Files**

Header File	PSE52	Description
aio.h	*	asynchronous input and output
assert.h	*	verify program assertion
complex.h	*	complex arithmetic (user-level only)
ctype.h	*	character types
dirent.h	*	format of directory entries
dlfcn.h	*	dynamic linking (user-level only)
errno.h	*	system error numbers
fcntl.h	*	file control options
fenv.h	*	floating-point environment (user-level only)
float.h	*	floating types (user-level only)
inttypes.h	*	fixed size integer types (user-level only)
iso646.h	*	alternative spellings (user-level only)
limits.h	*	implementation-defined constants
locale.h	*	category macros
math.h	*	mathematical declarations
mqueue.h	*	message queues
pthread.h	*	pthreads

Table 9-4 **POSIX Header Files** (cont'd)

Header File	PSE52	Description
sched.h	*	execution scheduling
search.h	*	search tables (user-level only)
semaphore.h	*	semaphores
setjmp.h	*	stack environment declarations
signal.h	*	signals
stdbool.h	*	boolean type and values (user-level only)
stddef.h	*	standard type definitions (user-level only)
stdint.h	*	integer types (user-level only)
stdio.h	*	standard buffered input/output
stdlib.h	*	standard library definitions
string.h	*	string operations
strings.h	*	string operations (user-level only)
sys/mman.h	*	memory management declarations
sys/resource.h	*	definitions for XSI resource operations
sys/select.h		select types (user-level only)
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 (user-level only)
sys/wait.h		declarations for waiting (user-level only)
tgmath.h		type-generic macros (user-level only)
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 (user-level only)
wctype.h		wide-character classification and mapping utilities (user-level only)

9.8 POSIX Namespace for RTP Applications

The POSIX.1 standard guarantees that the namespace of strictly conforming POSIX applications (user mode/RTP) 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.

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 9-4.

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 *9.7 POSIX Header Files*, p.185).

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 Diab 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.

9.9 POSIX Process Privileges

POSIX describes the concept of *appropriate privileges* associated with a process (RTP) 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( )**.

9.10 POSIX Process Support

VxWorks provides support for a user-mode process model with real-time processes (RTPs).

The following is 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 9-5 **POSIX Process Functions**

Function	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 functions, are provided with the **INCLUDE_RTP** component.

9.11 POSIX Clocks and Timers

For the kernel, VxWorks provides POSIX 1003.1b standard clock and timer interfaces. For RTP applications, 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 functions 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**.

POSIX CPU-Time Clocks for User-Mode Threads

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( )** function. The **CLOCK_THREAD_CPUTIME_ID** *clock_id* parameter refers to the calling thread's CPU-time clock.

POSIX Clock Functions

Table 9-6 lists the POSIX clock functions. The obsolete VxWorks-specific POSIX extension **clock_setres( )** is provided for backwards-compatibility purposes. For more information about clock functions, see the API reference for **clockLib**.

Table 9-6 **POSIX Clock Functions**

Function	Description
clock_getres( )	Kernel: Get the clock resolution (**CLOCK_REALTIME** and **CLOCK_MONOTONIC**). User: 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( )	Kernel: Get the current clock time (**CLOCK_REALTIME** and **CLOCK_MONOTONIC**). User: Get the current clock time (**CLOCK_REALTIME, CLOCK_MONOTONIC**, and thread CPU-time).
clock_settime( )	Kernel: Set the clock to a specified time for **CLOCK_REALTIME** (fails for **CLOCK_MONOTONIC**; not supported for a thread CPU-time clock in the kernel). User: Set the thread CPU-time clock (fails for **CLOCK_REALTIME** or **CLOCK_MONOTONIC**).
clock_nanosleep( )	User (only): Suspend the current pthread (or task) for a specified amount of time.
pthread_getcpuclockid( )	User (only): 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 functions for tasks to signal themselves at some time in the future. Functions 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( )** function (see *8.1 About Signals*, p. 152).

See Table 9-7 for a list of the POSIX timer functions. The VxWorks **timerLib** library includes a set of VxWorks-specific POSIX extensions: **timer_open( )**, **timer_close( )**, **timer_cancel( )**, **timer_connect( )**, and **timer_unlink( )**. These functions allow for an easier and more powerful use of POSIX timers on VxWorks. For more information, see the API reference for **timerLib**.

Table 9-7 **POSIX Timer Functions**

Function	Description
timer_create( )	Kernel: Allocate a timer using the specified clock for a timing base (**CLOCK_REALTIME** or **CLOCK_MONOTONIC**). User: 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 function 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.

NOTE: In VxWorks, a POSIX timer 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 timer 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 *7.18 Inter-Process Communication With Public Objects*, p.147.

Example 9-1 **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( )** function 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.

9.12 POSIX Asynchronous I/O

POSIX asynchronous I/O (AIO) functions are provided by the **aioPxLib** library.

The VxWorks AIO implementation meets the specification of the POSIX 1003.1 standard. For more information, see *11.8 Asynchronous Input/Output*, p.313.

9.13 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.

9.14 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 functions can be included in VxWorks to facilitate porting POSIX applications to VxWorks.

The kernel functions do not perform any action, as all pages are always kept in memory.

The user-level (RTP) functions do not perform any action except to validate the parameters passed, and if the addresses are passed, they are validated to ensure they are resident in memory. The functions do not perform any other action, as all pages are always kept in memory.

The POSIX page-locking functions are part of the memory management library, **mmanPxLib**, and are listed in Table 9-8.

Table 9-8 **POSIX Page-Locking Functions**

Function	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.

NOTE: The POSIX page-locking functions do not perform their intended operation in VxWorks. They are provided for application portability.

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.

9.15 POSIX Threads: Pthreads

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 *9.18 POSIX and VxWorks Scheduling*, p. 208.

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 function 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 system call handlers, see *19. Custom System Calls*.

POSIX Thread Stack Guard Zones For RTP Applications

In RTP (user-mode) applications, 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( )** function 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( )** function, 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( )** function 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 *Task Stack Guard Zones*, p.96.

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 functions* 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.

VxWorks-Specific Pthread Attributes: Name and Options

The VxWorks implementation of POSIX threads provides two additional pthread attributes (which are POSIX extensions)—pthread *name* and pthread *options*—as well as functions for accessing them.

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( )**

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( )**

Specifying Attributes when Creating Pthreads

The following examples create a pthread using the default attributes and using explicit attributes.

Example 9-2 **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 9-3 **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 9-4 **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 (stackaddr == 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);
```

POSIX Thread Creation and Management

VxWorks provides many POSIX thread functions. Table 9-9 lists a few that are directly relevant to pthread creation or execution. See the VxWorks API reference for information about the other functions, and more details about all of them.

Table 9-9 **POSIX Thread Functions**

Function	Description
pthread_create( )	Create a pthread.
pthread_cancel( )	Cancel the execution of a pthread
pthread_detach( )	Detach a running pthread so that it cannot be joined by another pthread.
pthread_join( )	Wait for a pthread to terminate.
pthread_getschedparam( )	Dynamically set value of scheduling priority attribute.
pthread_setschedparam( )	Dynamically set scheduling priority and policy parameter.
sched_get_priority_max( )	Get the maximum priority that a pthread can get.
sched_get_priority_min( )	Get the minimum priority that a pthread can get.
sched_rr_get_interval( )	Get the time quantum of execution of the round-robin policy.
sched_yield( )	Relinquishes the CPU.

POSIX Thread Attribute Access

The POSIX attribute-access functions are described in Table 9-10. The VxWorks-specific POSIX extension functions are described in section *VxWorks-Specific Pthread Attributes: Name and Options*, p.197.

Table 9-10 **POSIX Thread Attribute-Access Functions**

Function	Description
pthread_attr_getstacksize( )	Get value of the stack size attribute.
pthread_attr_setstacksize( )	Set the stack size attribute.
pthread_attr_getstackaddr( )	Get value of stack address attribute.
pthread_attr_setstackaddr( )	Set value of stack address attribute.
pthread_attr_getdetachstate( )	Get value of *detachstate* attribute (joinable or detached).
pthread_attr_setdetachstate( )	Set value of *detachstate* attribute (joinable or detached).

Table 9-10 **POSIX Thread Attribute-Access Functions** (cont'd)

Function	Description
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.

POSIX Thread Private Data

POSIX threads can store and access private data; that is, pthread-specific data. They use a *key* maintained for each pthread by the pthread library to access that data. A key corresponds to a location associated with the data. It is created by calling **pthread_key_create( )** and released by calling **pthread_key_delete( )**. The location is accessed by calling **pthread_getspecific( )** and **pthread_setspecific( )**. This location represents a pointer to the data, and not the data itself, so there is no limitation on the size and content of the data associated with a key.

The pthread library supports a maximum of 256 keys for all the pthreads in the kernel, and a maximum of 256 keys for all the pthreads in a given process.

The **pthread_key_create( )** function 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.

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 functions that can block the execution of the pthread for a lengthy period of time.

NOTE: While the **msync( )**, **fcntl( )**, and **tcdrain( )** functions 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 9-11.

Table 9-11 **Pthread Cancellation Points in VxWorks Libraries**

Library	Functions
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( )**
timerLib	**sleep( )**, **nanosleep( )**

Functions that can be used with cancellation points of pthreads are listed in Table 9-12.

Table 9-12 **Pthread Cancellation Functions**

Function	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.

1. Asynchronous cancellation is actually implemented with a special signal, **SIGCNCL**, which users should be careful not to block or to ignore.

Table 9-12 **Pthread Cancellation Functions** (cont'd)

Function	Description
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.

9.16 POSIX Thread Barriers

The VxWorks implementation of POSIX thread barriers complies with the POSIX standard IEEE Std 1003.1 (POSIX.1).

VxWorks supports POSIX thread barrier functions to assist with porting POSIX applications to VxWorks. Although VxWorks supports most of the barrier functions, the functions and attributes related to process shared barriers (**pthread_barrierattr_getpshared( )** and **pthread_barrierattr_setpshared( )**) are not, because VxWorks does not support the Thread Process-Shared Synchronization option.

VxWorks provides two new types: **pthread_barrier_t** and **pthread_barrierattr_t**.

To add POSIX thread barrier support, include the **INCLUDE_POSIX_PTHREADS** component. This component includes general POSIX threads support. It requires no additional configuration for POSIX thread barriers.

The following table lists the functions. Further details are in the pthreads documentation in *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition*.

Table 9-13**POSIX Thread Barrier Functions**

Function	Description
pthread_barrierattr_init( )	Initializes a barrier attributes object with a default value. The current implementation does not have implementation-specific attributes.
pthread_barrierattr_destroy( )	Destroys a barrier attributes object. The current implementation performs the necessary validation only; otherwise, it performs no operation. The POSIX.1 specification states that this function may set the object referenced by **attr** to an invalid value, which the VxWorks implementation does not do.
pthread_barrier_init( )	Initializes the final count. The count value must be greater than zero.

Table 9-13**POSIX Thread Barrier Functions** (cont'd)

Function	Description
pthread_barrier_wait( )	Updates the current count.
pthread_barrier_destroy( )	Deletes the barrier and releases any resources used by the barrier.

Using POSIX Thread Barriers

When you implement a thread barrier, all associated threads must reach the barrier and stop before proceeding further. For example, a thread barrier is useful if three threads need to initialize three separate, global variables and, once initialized, the threads use all three variables. Each thread could wait for the others at the barrier until the variables initialize.

According to the POSIX standard, when the required number of threads have called **pthread_barrier_wait( )** to specify the barrier, the call returns the constant **PTHREAD_BARRIER_SERIAL_THREAD** to one unspecified thread and zero to each of the remaining threads at which point all of the threads that reached the barrier are now eligible for execution. The call then resets the barrier to the state it had as a result of the most recent **pthread_barrier_init( )** call.

You can call the **pthread_barrier_wait( )** function up to the number of times specified by the **count** parameter in the **pthread_barrier_init( )** function, minus one, and expect the caller to be blocked. With the next caller, you can expect all blocked threads and the last thread to be eligible for execution; that is, they have passed the barrier and the barrier has closed behind them. If another thread calls the **pthread_barrier_wait( )** function, then that thread is blocked and the cycle begins again. The call returns an error (**EINVAL**) only if the barrier object is not initialized and, on VxWorks, when the synchronization primitives fail.

The following steps outline how to use thread barriers in VxWorks. Refer to the POSIX standard IEEE Std 1003.1 (POSIX.1) documentation and the API references for more information.

1. In a POSIX thread, call **pthread_barrier_init( )** to create the barrier, and initialize the current count to zero and the final count to the third argument. If the **pthread_barrierattr_t** pointer is not NULL, the POSIX thread library checks it to ensure it is initialized.
2. For every thread that needs to wait at the barrier (as defined by the **count** parameter), call **pthread_barrier_wait( )** to wait until the other threads have hit the barrier.
3. Delete the barrier by calling the **pthread_barrier_destroy( )** function. This function releases any resources, such as memory.

Example 9-5 **POSIX Thread Barrier Example Code**

```
STATUS appInit (void)
    {
…
    pthread_t thread1, thread2, thread3;
    pthread_barrier_t myBarrier;

    /* Turn this task into a POSIX thread */

    pthread_self();
    /* Initializes the barrier "myBarrier" to wait for three threads. */

    pthread_barrier_init (&myBarrier, NULL, 3);

    /* Create three threads. */

    ret = pthread_create (&thread1, NULL, &doWork, (void *)&myBarrier);
    ret = pthread_create (&thread2 ,NULL, &doWork, (void *)&myBarrier);
    ret = pthread_create (&thread3, NULL, &doWork, (void *)&myBarrier);

    pthread_barrier_destroy (&myBarrier);
…
    pthread_exit ((void *)OK);
    }

void * doWork
    (
    void * pMyBarrier
    )
    {
…
    pthread_barrier_wait (pMyBarrier);
…
    pthread_exit (NULL);
    }
```

9.17 POSIX Pthread Mutexes and Condition Variables

Kernel pthread mutexes (mutual exclusion variables) and condition variables provide compliance with the POSIX 1003.1c standard.

Like POSIX threads, mutexes and condition variables have *attributes* associated with them. Mutex attributes are held in a data type called **pthread_mutexattr_t**, which contains two attributes, **protocol** and **prioceiling**. The functions used to manage these attributes are described below. For more information about these and other mutex functions, see the API reference for **pthreadLib**.

User-level 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 functions used to manage these attributes are described below. For more information about these and other mutex functions, see the VxWorks API reference for **pthreadLib**.

Thread Mutexes

The functions that can be used to act directly on a mutex object and on the mutex attribute object are listed in Table 9-14 and Table 9-15 (respectively).

Table 9-14 **POSIX Functions Acting on a Mutex Object**

Function	Description
pthread_mutex_destroy( )	Destroy a mutex.
pthread_mutex_init( )	Initialize a mutex.
pthread_mutex_getprioceiling( )	Get the priority ceiling of a mutex.
pthread_mutex_setprioceiling( )	Set the priority ceiling of a mutex.
pthread_mutex_lock( )	Lock a mutex.
pthread_mutex_trylock( )	Check and lock a mutex if available.
pthread_mutex_unlock( )	Unlock a mutex.
pthread_mutex_timedlock( )	Lock a mutex with timeout. User-level only.

For information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p. 149.

Table 9-15 **POSIX Functions Acting on a Mutex Attribute Object**

Function	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. User-level only.
pthread_mutexattr_settype( )	Set a mutex **type** attribute. User-level only.

User-Level Type Mutex Attribute

The user-level **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 *7.8 Mutual-Exclusion Semaphores*, p.123).

- Attribute Name: **protocol**
- Possible Values: **PTHREAD_PRIO_NONE**, **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**
- Access Functions: **pthread_mutexattr_getprotocol( )** and **pthread_mutexattr_setprotocol( )**

A user-level mutex is created by default 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 kernel **PTHREAD_PRIO_INHERIT** option is the default value of the protocol attribute for pthreads created in the kernel (unlike pthreads created in processes, for which the default is **PTHREAD_PRIO_NONE**).

The **PTHREAD_PRIO_INHERIT** value is used to create a mutex with priority inheritance—and is equivalent to the association of **SEM_Q_PRIORITY** and **SEM_INVERSION_SAFE** options used with **semMCreate( )**. A pthread owning a mutex variable created with the **PTHREAD_PRIO_INHERIT** value inherits the priority of any higher-priority pthread waiting for the mutex and executes at this elevated priority until it releases the mutex, at which points it returns to its original priority.

Because it might not be desirable to elevate a lower-priority pthread to a priority above a certain level, POSIX defines the notion of priority ceiling, described below. Mutual-exclusion variables created with *priority protection* use the **PTHREAD_PRIO_PROTECT** value.

Priority Ceiling Mutex Attribute

The **prioceiling** attribute is the POSIX priority ceiling for mutex variables created with the **protocol** attribute set to **PTHREAD_PRIO_PROTECT**.

- Attribute Name: **prioceiling**
- Possible Values: any valid (POSIX) priority value (0-255, with zero being the lowest).

- Access Functions: **pthread_mutexattr_getprioceiling( )** and **pthread_mutexattr_setprioceiling( )**
- Dynamic Access Functions: **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 *POSIX and VxWorks Priority Numbering*, p.210.

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.

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 functions that can be used to act directly on a condition variable and on the condition variable attribute object are listed in Table 9-14 and Table 9-15 (respectively).

Table 9-16 **POSIX Functions Acting on a Condition Variable Object**

Function	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.

For information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p.149.

Table 9-17 **POSIX Functions Acting on a Condition Variable Attribute Object**

Function	Description
pthread_condattr_destroy( )	Destroy condition variable attributes object.
pthread_condattr_init( )	Initialize condition variable attributes object.

Table 9-17 **POSIX Functions Acting on a Condition Variable Attribute Object** (cont'd)

Function	Description
pthread_condattr_getclock( )	Get the clock selection condition variable attribute. User-level only.
pthread_condattr_setclock( )	Set the clock selection condition variable attribute. User-level only.

User-Level Clock Selection

The only attribute supported for user-level condition variable objects is the clock ID. By default the **pthread_cond_timedwait( )** function uses the **CLOCK_REALTIME** clock. The clock type can be changed with the **pthread_condattr_setclock( )** function. The accepted clock IDs are **CLOCK_REALTIME** (the default) and **CLOCK_MONOTONIC**. For information about POSIX clocks, see *9.11 POSIX Clocks and Timers*, p.190.

9.18 POSIX and VxWorks Scheduling

Neither the default (traditional/native) VxWorks scheduler nor the VxWorks POSIX thread scheduler can be used to schedule processes (RTPs). The only entities that can be scheduled 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.

(The Safety Profile provides an additional scheduler that allows for scheduling RTPs in time windows; see *15.11 RTP Time Partition Scheduling*, p.388.)

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 9-18 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.214.)

- 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 9-18 is discussed in detail in subsequent sections.

Table 9-18 **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. The traditional VxWorks scheduler cannot ensure behavioral compliance with the POSIX 1 standard

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.

- The POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *POSIX and VxWorks Priority Numbering*, p.210.
- 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.

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.

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 user-mode pthreads (running in RTPs). 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( )** function. 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( )** function 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.

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 *Default Scheduling Policy*, p.210), and concurrent policies cannot be applied to individual pthreads. For general information about the traditional scheduler and how it works with tasks, see *6.5 Task Scheduling*, p.85.

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.213.
- **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 *POSIX Threads Scheduler*, p.211.

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 *POSIX and VxWorks Priority Numbering*, p.210.

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 *6.5 Task Scheduling*, p.85 for information about the traditional scheduler and how it works with VxWorks tasks, and *VxWorks Traditional Scheduler*, p.210 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.214).
- **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.

By default, **INCLUDE_POSIX_PTHREAD_SCHEDULER** includes the **INCLUDE_PX_SCHED_DEF_POLICIES** component, which supports the base POSIX thread's scheduling policies (the **SCHED_FIFO**, **SCHED_RR**, and **SCHED_OTHER**).

To use the sporadic scheduling policy, the **INCLUDE_PX_SCHED_SPORADIC_POLICY** component must be included instead of **INCLUDE_PX_SCHED_DEF_POLICIES**.

Note that when the configuration is done via the Workbench including one or the other of the scheduling policy components is done via a selection.

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( )** function—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( )** function—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.87.

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).

POSIX Scheduling Functions

The POSIX 1003.1b scheduling functions provided by the **schedPxLib** library for VxWorks are described in Table 9-19.

Table 9-19 **POSIX Scheduling Functions**

Function	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 functions, see the **schedPxLib** API reference.

NOTE: Several kernel scheduling functions that were provided with **schedPxLib** for VxWorks 5.*x* and early versions of VxWorks 6.*x* are not POSIX compliant, and are maintained only for backward compatibility in the kernel. The use of these functions is deprecated: **sched_setparam( )**, **sched_getparam( )**, **sched_setscheduler( )**, and **sched_getscheduler( )**.

The native VxWorks functions **taskPrioritySet( )** and **taskPriorityGet( )** should be used for task priorities. The POSIX functions **pthread_setschedparam( )** and **pthread_getschedparam( )** should be used for pthread priorities.

For information about changing the default system scheduling policy, see *Default Scheduling Policy*, p.210. For information about concurrent scheduling policies, see *POSIX Threads Scheduler*, p.211.

Note that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *POSIX and VxWorks Priority Numbering*, p.210.

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.

Getting Scheduling Parameters: Priority Limits and Time Slice

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

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

Example 9-6 **Getting the POSIX Round-Robin Time Slice in the Kernel**

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

/* includes */

#include <vxWorks.h>
#include <sched.h>

STATUS rrgetintervalTest (void)
    {
    struct timespec slice;

    /* turn on round robin */

    kernelTimeSlice (30);

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

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

User-mode tasks and pthreads in RTPs can use **sched_rr_get_interval( )** to determine the length of the current time-slice interval. This function 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 function is being scheduled with the **SCHED_RR** policy. To make this determination, a pthread must use the **pthread_getschedparam( )** function.

Example 9-7 **Getting the POSIX Round-Robin Time Slice in User Space**

```
/*
 * 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);
}
```

9.19 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 functions for creating, opening, and destroying both named and unnamed semaphores.

When opening a named semaphore, you assign a symbolic name, which the other named-semaphore functions accept as an argument.

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

The POSIX semaphore functions provided by **semPxLib** are shown in Table 9-20.

Table 9-20 **POSIX Semaphore Functions**

Function	Description
sem_init( )	Initializes an unnamed semaphore.
sem_destroy( )	Destroys an unnamed semaphore.
sem_open( )	Initializes/opens a named semaphore.
sem_close( )	Closes a named semaphore.
sem_unlink( )	Removes a named semaphore.
sem_wait( )	Lock a semaphore.
sem_trywait( )	Lock a semaphore only if it is not already locked.
sem_post( )	Unlock a semaphore.
sem_getvalue( )	Get the value of a semaphore.
sem_timedwait( )	Lock a semaphore with a timeout. The timeout is based on the **CLOCK_REALTIME** clock.

For information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p.149.

Note that the behavior of the user-level **sem_open( )** function complies with the POSIX specification, and thus differs from the kernel version of the function. 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-level version of **sem_open( )** returns the same ID for the same named semaphore when called multiple times.

To include the kernel **semPxLib** library semaphore functions in the system, configure VxWorks with the **INCLUDE_POSIX_SEM** component.

VxWorks provides **semPxLibInit( )**, a non-POSIX (kernel-only) function 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.

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 functions for locking, unlocking, and getting the value of semaphores are used for both named and unnamed semaphores.

The functions **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 functions **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.

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 function, **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-mode application, unnamed semaphores can be accessed only by the tasks belonging to the process executing the application.

NOTE: In VxWorks, a POSIX semaphore 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 semaphore 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 *7.18 Inter-Process Communication With Public Objects*, p.147.

Example 9-8 **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 function 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 */
}
```

Using Named Semaphores

The **sem_open( )** function 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 9-21.

Table 9-21 **Possible Outcomes of Calling sem_open( )**

Flag Settings	If Semaphore Exists	If Semaphore Does Not Exist
None	Semaphore is opened.	Function fails.
O_CREAT	Semaphore is opened.	Semaphore is created.
O_CREAT and **O_EXCL**	Function fails.	Semaphore is created.
O_EXCL	Function fails.	Function 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. (Note that the **show( )** function is not a POSIX function, nor is it meant to be used programmatically.

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
```

The **show( )** function takes the semaphore ID as the argument.

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

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

NOTE: POSIX named semaphores may be shared between processes only if their names start with a **/** (forward slash) character. They are otherwise private to the process in which they were created, and cannot be accessed from another process. See *7.18 Inter-Process Communication With Public Objects*, p.147.

Example 9-9 **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 function 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 function 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;
        }
    }
```

9.20 POSIX Message Queues

The POSIX message queue functions are provided by **mqPxLib**.

Table 9-22 **POSIX Message Queue Functions**

Function	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. User-level only.
mq_timedreceive( )	Gets a message from a queue, with a timeout. User-level only.

Note that there are behavioral differences between the kernel and user-level versions of **mq_open( )**. The kernel version allows for creation of a message queue for any permission specified by the *oflags* parameter. The user-space version complies with the POSIX PSE52 profile, so that after the first call, any subsequent calls in the same process are only allowed if an equivalent or lower permission is specified.

For more information about the VxWorks message queue library, see the **msgQLib** API reference.

For information about user-level native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p. 149.

Table 9-23 describes the permissions that are allowed with the user-level APIs.

Table 9-23 **User-Level 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 kernel, the VxWorks initialization function **mqPxLibInit( )** initializes the kernel's POSIX message queue library (this is a kernel-only function). It is called automatically at boot time when the **INCLUDE_POSIX_MQ** component is part of the system.

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 9-24.

Table 9-24 **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_timedsend( )** (user-space only)

For information about native VxWorks API timeouts and POSIX API timeouts, see *7.19 About VxWorks API Timeout Parameters*, p. 149.

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 9-10 **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);
    }
```

Displaying Kernel Message Queue Attributes

The kernel function **mqPxShow( )** can be used to display information about POSIX message queues, as illustrated below.

```
-> mq_open ("mymq4", 0x4201, 0)
value = 8380448 = 0x7fe020
-> mqPxShow 0x7fe020
Message queue name              : mymq4
No. of messages in queue        : 0
Maximum no. of messages         : 16
Maximum message size            : 16
Flags                           : O_WRONLY  O_NONBLOCK   (0x4001      )
```

Note that the non-POSIX **show( )** function is used with VxWorks message queues, and provides different output.

```
-> show myMsgQId
Message Queue Id    : 0x7fd0b0
Task Queuing        : FIFO
Message Byte Len    : 256
Messages Max        : 10
Messages Queued     : 0
Receivers Blocked   : 0
Send timeouts       : 0
Receive timeouts    : 0
Options             : 0x0       MSG_Q_FIFO
```

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 *7.18 Inter-Process Communication With Public Objects*, p.147.

Example 9-11 **POSIX Message Queues**

```
/*
 * In this example, the mqExInit() function 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 function 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 function 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 function 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");
    }
```

Notification of Message Arrival

A pthread (or task) can use the **mq_notify( )** function 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 9-12 **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 function 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 function 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 function 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);
    }
    }
```

9.21 POSIX Pipes

POSIX pipes and FIFOs—also known as unnamed pipes and named pipes, respectively—provide a unidirectional data channel that can be used for intertask and interprocess communication.

Unnamed pipe can be used for inter-process communication between parent and child RTPs. The file descriptors opened by a parent RTP are inherited by its child RTPs. File descriptor numbers can also be passed with **rtpSpawn( )** using the **argv** argument when the child process is spawned. Since file descriptors opened by the kernel are not inherited by RTPs that are spawned by the kernel, unnamed pipes cannot be used between the kernel and an RTP.

FIFOs can be used for used for inter-process communication between real-time processes (RTPs), or the kernel and an RTP.

Both unnamed pipes and FIFOs can also be used for intertask communication within the same RTP or within the kernel.

POSIX Compliance

The VxWorks implementation of POSIX pipes and FIFOs is mostly compliant with the Open Group Base Specifications. The deviations from the specifications are as follows:

- A FIFO special file must be created on a specific path (defined by the **FIFOS_DEV_NAME** configuration parameter), which is restricted to a flat directory. A subdirectory cannot be created below **FIFOS_DEV_NAME**. The FIFO special file cannot be moved from the path or renamed. The path argument used with **mkfifo( )** must point to the path on the **FIFOS_DEV_NAME** device.
- A FIFO special file is not a persistent file, as it is not created on a physically mounted I/O device. FIFO special files are destroyed when the system reboots and must be recreated after rebooting, before opening. A FIFO special file does not require a real file system for I/O operations.
- For **pipe2( )**, only the access control flag **O_NONBLOCK** is supported. The **O_CLOEXEC** flag is not supported as VxWorks does not support the **FD_CLOEXEC** (close-on-exec) flag.
- The file descriptors created and opened by the kernel are not inherited by RTPs that are spawned by the kernel. This is a limitation of the VxWorks I/O system.

Comparison of POSIX and VxWorks Pipes

The differences between POSIX and VxWorks pipes are summarized in Table 9-25.

Table 9-25 **POSIX and VxWorks Pipes**

VxWorks Pipes	POSIX Pipes and FIFOs
VxWorks pipes use the VxWorks message queue facility to do the buffering and delivering of messages. The VxWorks pipes facility provides access to the message queue facility through the I/O system.	POSIX pipes use a ring buffer to read and write data. The POSIX pipes facility provides access to the ring buffer facility through the I/O system.
VxWorks pipes are named (with I/O device names).	Unnamed POSIX pipes are not named. FIFOs are named (with I/O device names).
VxWorks pipes are: created with **pipeDevCreate( )**, opened with **open( )**, and deleted with **unlink( )**.	Unnamed POSIX pipes are: created and opened with **pipe( )** or **pipe2( )**, and deleted with **close( )**. FIFOs are: created with **mkfifo( )**, opened with **open( )**, and deleted with **unlink( )**.

Table 9-25 **POSIX and VxWorks Pipes**

VxWorks Pipes	POSIX Pipes and FIFOs
The size of the buffer for writing is set when a VxWorks pipe is created, with a parameter to **pipeDevCreate( )**.	If the POSIX pipe is created with the **O_NONBLOCK** flag set, the size of the ring buffer for writing is fixed—the maximum size (in bytes) of the buffer available for writing is set (statically) with the **PX_PIPE_BUFFER_SIZE** configuration parameter, and the available amount is then **PX_PIPE_BUFFER_SIZE** minus the size of any data remaining in the pipe.
	If the pipe is created with **O_NONBLOCK** flag cleared, there is no limitation to the size of the buffer for writing.
The maximum number of file descriptors for VxWorks pipes is set (statically) with the **PIPE_MAX_OPEN_FDS** configuration parameter.	In the kernel, any number of POSIX pipes can be opened as long as the total number of file descriptors for all the I/O devices in the kernel does not exceed the value of the **NUM_FILES** configurable parameter.
	In a given RTP, the total number of file descriptors for all the I/O devices cannot exceed the value of the **RTP_FD_NUM_MAX** configuration parameter.
ISRs can write to a VxWorks pipe, but cannot read from a VxWorks pipe.	ISRs cannot read from or write to POSIX pipes and FIFOs.
VxWorks pipes do not comply with the Open Group Base Specifications.	**pipe( )** and **mkfifo( )** mostly comply with the Open Group Base Specifications.

Configuring the Object Device Name and Ring Buffer Size

The **INCLUDE_POSIX_PIPES** component provides the following parameters for static configuration of POSIX pipes and FIFOs:

FIFOS_DEV_NAME
The name of POSIX pipes and FIFOs object device. The default is **/fifos**.

PX_PIPE_BUFFER_SIZE
The ring buffer size for each POSIX pipe and FIFO. The default size is 4096 bytes. If **PX_PIPE_BUFFER_SIZE** is set to a size less than **_POSIX_PIPE_BUF** (512 bytes, as defined in **limits.h**), then a ring buffer size equal to **_POSIX_PIPE_BUF** is used regardless.

Creating POSIX Pipes

The functions provided for creating unnamed POSIX pipes and FIFOs are described in Table 9-26. The user and kernel unnamed pipe functions are identical, except for their use in different memory spaces.

Table 9-26 **Functions for Creating POSIX Named Pipes and FIFOs**

Function	Description
pipe( )	Creates and opens an unnamed pipe. The **fcntl( )** function can be used to set the **O_NONBLOCK** flag.
pipe2( )	Creates and opens an unnamed pipe like **pipe( )**, but takes an additional argument that sets or clears the **O_NONBLOCK** file status flag.
mkfifo( )	Creates a FIFO with the name that specified with the call (kernel function only). FIFOs are not opened when created; the **open( )** function must be used to do so.

Pipes and FIFOs are created on the object device named with the **FIFOS_DEV_NAME** configuration parameter (the default is **/fifos**).

For **pipe2( )**, only the access control flag **O_NONBLOCK** is supported. The **O_CLOEXEC** flag is not supported as VxWorks does not support the **FD_CLOEXEC** (close-on-exec) flag.

For more information, see the **pxPipeDrv** and **ioLib** API reference entries.

Special Properties of FIFOs

- The pipe device, which is named with the **FIFOS_DEV_NAME** configuration parameter, is restricted to a flat directory.
- A sub-directory cannot be created on the pipe device.
- The *path* argument specified with the **mkfifo( )** call must point to the path on the pipe device.
- The FIFO special file cannot be moved from the specified path or renamed.
- A FIFO special file is not a persistent file since it is not created on a physically mounted I/O device.
- FIFO special files are destroyed at system reboot, and therefore must be recreated before opening after a reboot.
- FIFO special files do not require a real file system for I/O operations.

Reading From Pipes

For FIFOs, the read end of the pipe is opened using the **O_RDONLY** flag (the write end is opened using the **O_WRONLY** flag).

In other respects, **read( )** calls to a POSIX pipe or FIFO are handled in the same way as a regular file, but with the following exceptions.

When attempting to read from an empty POSIX pipe or FIFO:

- If no task has the pipe open for writing, **read( )** returns 0 to indicate end-of-file.

- One or more tasks have the pipe open for writing and **O_NONBLOCK** is set, a **read( )** call returns -1 and set errno to **EAGAIN**.
- One or more tasks have the pipe open for writing and **O_NONBLOCK** is clear, a **read( ) call** blocks the calling task until some data is written or the pipe is closed by all tasks that had the pipe open for writing.

When attempting to read from a POSIX pipe or FIFO that has fewer bytes than being requested to be read:

- If a **read( )** operation is interrupted by a signal before it reads any data, it returns -1 with errno set to **EINTR**.
- If a **read( )** operation is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

Writing to Pipes

For FIFOs, the write end of the pipe is opened using the **O_WRONLY** flag (the read end is opened using the **O_RDONLY** flag).

In other respects, **write( )** calls to a POSIX pipe or FIFO are handled in the same way as a regular file, but with the following exceptions:

- There is no file offset associated with a POSIX pipe, hence each write request is appended to the end of the pipe.
- Write requests of **_POSIX_PIPE_BUF** size (512 bytes, as defined in **limits.h**) or less are not interleaved with data from other tasks doing write operations on the same pipe.
- Write requests greater than **_POSIX_PIPE_BUF** size bytes may have data interleaved, with writes by other tasks, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is clear, write request may cause the task to block, but on normal completion it returns the number of bytes requested with the **write( )** call.
- If the **O_NONBLOCK** flag is set, a write request for **_POSIX_PIPE_BUF** or fewer bytes has the following effects:
  - If there is sufficient space available in the pipe, the **write( )** call transfers all the data and return the number of bytes requested.
  - Otherwise, the **write( )** call transfers no data and returns -1 with errno set to **EAGAIN**.
- If the **O_NONBLOCK** flag is set, a **write( )** call with more than **_POSIX_PIPE_BUF** size bytes has the following effects:
  - When at least one byte can be written, **write( )** transfers what it can and returns the number of bytes written. When all data previously written to the pipe is read, it transfers at least **_POSIX_PIPE_BUF** size bytes.
  - When no data can be written, **write( )** transfers no data, and return -1 with errno set to **EAGAIN**.

I/O Control Functions

The **ioctl( )** functions described in Table 9-27 can be used with POSIX pipes and FIFOs. These **ioctl( )** function codes are defined in the header file **ioLib.h**.

Table 9-27 **I/O Control Function**

Function	Description
FIOGETNAME	Get file name.
FIOFSTATGET	Get file status information.
FIOFCNTL	Get or set the file flag.
FIOPATHCONF	Get the current value of a configurable limit.
FIOCHMOD	Change the permissions mode of a file.
FIOACCESS	Determine accessibility of a file.
FIONREAD	Get number of bytes in the queue.
FIONWRITE	Get number of bytes available to write.
FIOFLUSH	Discard all outstanding messages.
FIOSELECT	Select wake up process on I/O.
FIOUNSELECT	Deselect wake up process on I/O.

Deleting Pipes

Unnamed POSIX pipes are deleted with **close( )**. FIFOs are deleted with **unlink( )**. When all file descriptors associated with a pipe or FIFO are closed, any data remaining in the pipe or FIFO is discarded.

9.22 POSIX Trace

VxWorks provide support for the user-level POSIX trace facility for RTP applications. 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**. For more information, see the *VxWorks Kernel Shell User's Guide*.

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 **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( )** function 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( )** function.

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 functions 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 functions used for the controller, generator, and analyzer roles are described in Table 9-28.

Table 9-28   **POSIX Trace Functions**

Function	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

Table 9-28 **POSIX Trace Functions** (cont'd)

Function	Description	Role
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 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

This is the trace record that the code example produces.

```
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:
```

9.23 POSIX System Logging

The VxWorks **syslogLib** library provides facilities for tracking and logging system messages.

The library is POSIX conformant (POSIX.1-2008) except for the **%m** conversion specification. Support is provided for both the kernel and user mode (RTP) applications, with the exception of **syslogFdSet( )**, which is available only in the kernel.

The **syslogLib** facility can be configured to log messages before it is initialized. These messages are stored in static memory until initialization is complete and the system logger has started (**tLogTask**). As the POSIX clocks facility has also not been initialized when log messages are written to static memory, the timestamp assigned to these messages is not accurate (it starts from 1970-01-01T00:00:00.000000Z).

VxWorks Configuration

The VSB features required for system logging are also required for basic kernel and RTP support, and therefore do not need selection for your VSB project.

The **INCLUDE_SYSLOG** VIP component provides the system logging facility.

The **LOG_TASK_PRIORITY** parameter defines the priority with which the logging task **tLogTask** is spawned. The default value is zero.

The **NUM_EARLY_SYSLOG_MSGS** parameter defines the number of messages that can be logged before the logging facility is initialized. The default value is zero.

The maximum number of messages is defined by the **MAX_LOG_MSGS** parameter of the **INCLUDE_LOGGING** component.

syslogLib APIs

Function	Description
openlog( )	Set process attributes.
closelog( )	Unset process attributes.
setlogmask( )	Set log priority mask.
syslog( )	Send a message to the system logger.
vsyslog( )	Send a string formatted message with a variable argument list to the system logger (**tLogTask**).
syslogFdSet( )	Set the logging file descriptor. Kernel-only function.

For more information as well as code examples, see the **syslogLib** API reference entries (for kernel and user APIs).

10
Memory Management

10.1 VxWorks 6.9-Compatible Memory Model for 32-Bit BSPs

To simplify migration to VxWorks 7, the legacy VxWorks 6.9 32-bit memory model is supported for VxWorks 6.9-compatible 32-bit BSPs.

Using this memory model requires specifying 6.9-compatible mode when you create your VSB project. With **wrtool**, use the **-compat69** option. With Workbench, in the Project Setup dialog, select
a VxWorks 6.9 compatible board support package.

For information about the 6.9 memory model for 32-bit VxWorks, as well as the associated kernel heap and memory partition management facilities, see the 6.9 *VxWorks Kernel Programmer's Guide: Memory Management* and the *VxWorks Application Programmer's Guide: Memory Management*. For information about RTP memory configuration options for 32-bit VxWorks, see the 6.9 *VxWorks Application Programmer's Guide: Real Time Processes*.

NOTE: VxWorks 6.9-compatibility does not support for MMU-less systems with RTPs. Support for MMU-less kernel-only systems is provided by the **MMULESS_KERNEL** VSB option.

10.2 VxWorks 7 Memory Model

VxWorks provides memory management facilities for all code that executes in the kernel, as well as for applications that execute as real-time processes (RTPs). The VxWorks 7 memory model is the same for 32-bit and 64-bit CPUs.

The kernel context is not identity mapped, which means that virtual memory addresses are not equal to physical addresses. Full MMU support is therefore always required for this memory model. Virtual memory is split into regions, each with a well-defined purpose and corresponding allocation facilities.

NOTE: VxWorks 7 does not support MMU-less systems with RTPs. Support for MMU-less kernel-only systems is provided by the **MMULESS_KERNEL** VSB option.

The VxWorks 7 memory model has been designed to provide:

- Improved support for systems with large amounts of RAM.
- Support for systems with discontinuous blocks of RAM,
- Faster, more efficient system call memory validation.
- Reduced fragmentation of the virtual address space.
- Dynamic mapping of memory and I/O space as needed (instead of using static configuration data).
- Facilitation of shared library development using the industry-standard ABI, which requires standard overlapped address space management based on a predefined virtual memory layout.

The most significant consequence of these design considerations is that, in contrast to the VxWorks 6.9-compatible 32-bit legacy model, the kernel context is not identity mapped. This means that virtual memory address are not equal to physical addresses. It also means that virtual address space has a well-defined structure. Full MMU support is always required for this memory model.

While on 6.9 legacy 32-bit systems the virtual address space is fragmented around identity-mapped kernel blocks, in the VxWorks 7 model the virtual memory is split into regions, each with a well-defined purpose and corresponding allocation facilities.

NOTE: In contrast to legacy 32-bit VxWorks, the virtual memory regions of VxWorks 7 are fixed. The operating system performs virtual memory assignments from those regions and creates page mappings to physical memory as needed.

Displaying Information About Memory Layout

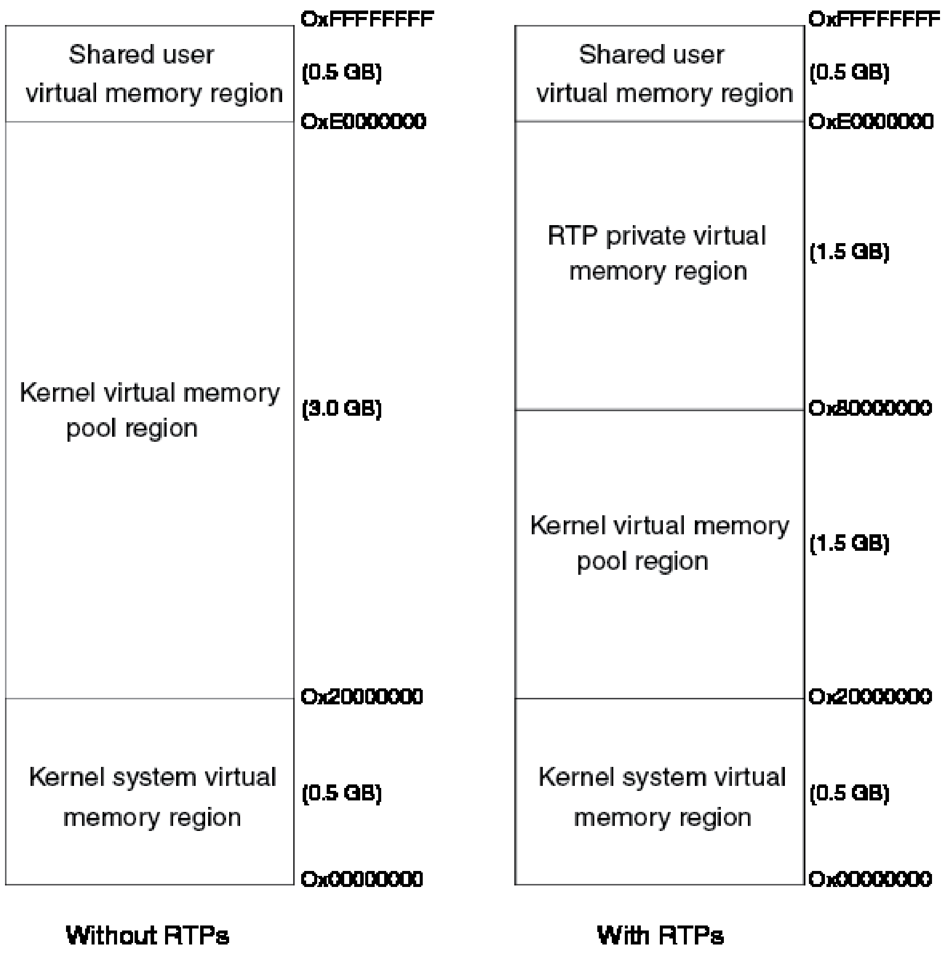
The shell's **adrSpaceShow( )** show function (for the C interpreter) or the **adrsp info** command (for the command interpreter) can be used to display an overview of the address space usage at time of the call. These are included in the kernel with the **INCLUDE_ADR_SPACE_SHOW** and **INCLUDE_ADR_SPACE_SHELL_CMD** components, respectively.

Virtual Memory Regions

VxWorks defines a number of virtual memory regions. These regions, with addresses and sizes specific to the processor architecture, have well-defined purposes. The virtual memory regions are illustrated in Figure 10-1. The illustration is intended to serve as a conceptual aid, and is neither definitive nor to scale.

Information about the location and size of each region can obtained using the **adrSpaceShow( )** function.

Figure 10-1 **Virtual Memory Regions (for PPC and ARM)**


Shared user virtual memory region
0xFFFFFFFF
(0.5 GB)
0xE0000000
Kernel virtual memory pool region
(3.0 GB)
0x20000000
Kernel system virtual memory region
(0.5 GB)
0x00000000
Without RTPs
Shared user virtual memory region
0xFFFFFFFF
(0.5 GB)
0xE0000000
RTP private virtual memory region
(1.5 GB)
0x80000000
Kernel virtual memory pool region
(1.5 GB)
0x20000000
Kernel system virtual memory region
(0.5 GB)
0x00000000
With RTPs

Kernel System Virtual Memory Region
: The kernel system virtual memory region contains the kernel system memory. It includes the kernel system memory, where the kernel image (text, data and bss sections), and the kernel proximity heap, among other areas, are located.

Kernel Virtual Memory Pool Region
: The kernel virtual memory pool region is used for dynamically managed mappings in the kernel. This region is used to allocate virtual memory as needed for creation and expansion of the kernel application, memory-mapped devices, DMA memory, user-reserved memory and persistent memory. Detailed information about the kernel virtual memory pool region can be obtained using the **adrSpaceShow( )** function.

Kernel Reserved Memory Region
: This kernel region is reserved for VxWorks internal usage, such as for the management of MMU page table structures.

Shared User Virtual Memory Region
: The shared user virtual memory region is used for allocating virtual memory for shared mappings such as shared data regions, shared libraries, memory mapped with **mmap( )** using the **MAP_SHARED** option. Detailed information about the shared user virtual memory region can be obtained using the **adrSpaceShow( )** function.

RTP Private Virtual Memory Region
: This RTP private virtual memory region used for the creation of RTP private mappings: the text and data segment, RTP heap, and memory mapped with **mmap( )** using the **MAP_PRIVATE** option. All RTPs in the system have access to the entire RTP private virtual memory region. In other words, RTP support always uses overlapped address space management.

Global RAM Pool

The global RAM pool is an internal allocation facility for dynamic allocation of RAM. VxWorks uses this for the creation and expansion of the kernel common heap, RTP private memory and shared memory. The global RAM pool also accounts for memory used for the VxWorks kernel image, user-reserved memory, persistent memory, DMA32 heap, and so on.

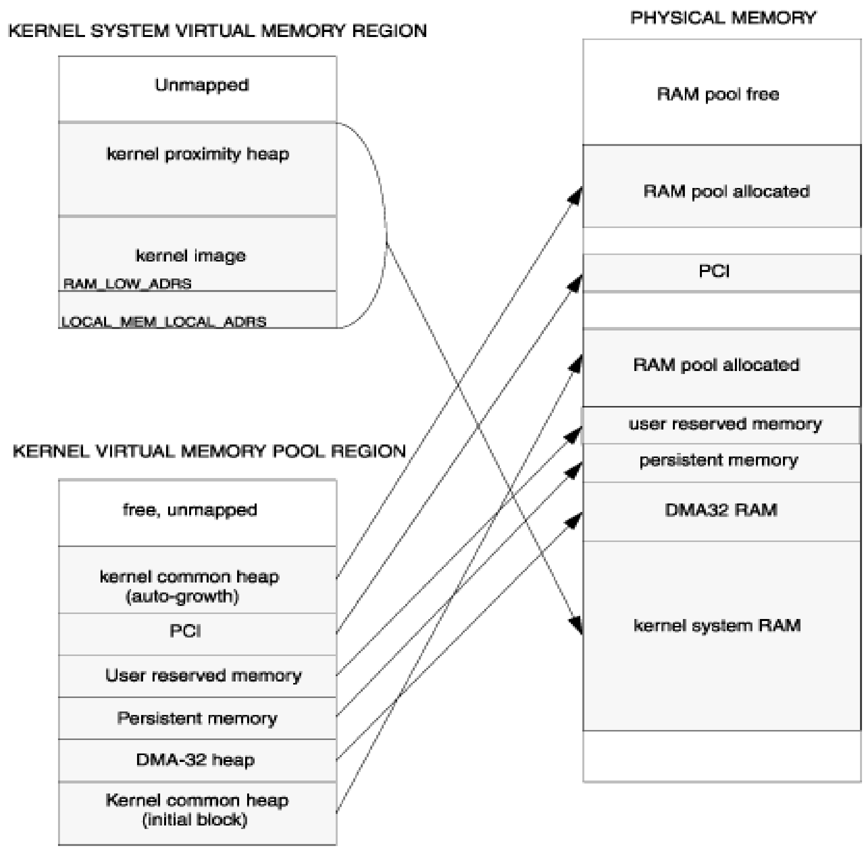
The global RAM pool is initialized based on the memory configuration provided by the BSP. Every RAM segment of all types is added to the pool. The initial state of the segments is set based on the usage: kernel system RAM, user-reserved memory, persistent memory and DMA32 memory is implicitly set non-free. Other RAM segments are initially added free, and dynamically allocated as needed, for example for kernel common heap creation, RTPs, and so on.

The content of the RAM pool can be displayed using the **adrSpaceShow( )** function.

Kernel Memory Map

Figure 10-2 provides an illustration of the kernel memory map (excluding the reserved region). The illustration is intended to serve as a conceptual aid, and is neither definitive nor to scale.

Figure 10-2 **Virtual and Physical Memory Maps**



Kernel System Memory

Kernel system memory is located in the kernel system memory region, and contains the kernel code segments, interrupt stack, memory for the initial VxWorks task, the kernel proximity heap, and the special storage areas such as the boot parameter area and exception message area. The kernel system memory is mapped as one virtually and physically contiguous block of RAM. The virtual memory start address is defined as **LOCAL_MEM_LOCAL_ADRS**. The size is the sum of the items it holds; that is, the kernel image, kernel proximity heap, and so on. For more information about the kernel proximity heap see *Kernel Proximity Heap*, p. 251. The RAM for the kernel system memory is configured by the BSP.

Kernel Common Heap

The kernel common heap is the memory partition used by the kernel and kernel applications for dynamic memory allocation. Physical memory is allocated from the global RAM pool, virtual memory is allocated from the kernel virtual memory pool region. For more information about the kernel common heap see *Kernel Common Heap*, p. 250.

DMA32 Heap

The DMA32 heap is a memory partition used by drivers for devices that do not have the capability to address more than 4 GB of physical address space (such as 32-bit PCI devices). The virtual address is dynamically allocated from the kernel virtual memory pool region. The physical address and size of the DMA32 heap is provided by the BSP. For more information about the DMA32 heap see *DMA32 Heap*, p.251.

User-Reserved Memory
: User-reserved memory is an optional portion of the RAM that is mapped—but not directly managed—by the kernel. It is managed by applications running in the kernel. The virtual memory for user-reserved memory is allocated from the kernel virtual memory pool region. The **userReservedMem( )** function provides information about the user-reserved memory (start address and size). For information about configuration, see *User-Reserved Memory Configuration*, p.250.

Persistent Memory
: Persistent memory is an optional portion of the RAM that is not cleared on warm reboot. The virtual memory for persistent memory is allocated from the kernel virtual memory pool region. Persistent memory is managed with the **pmLib** API. Persistent memory is used by kernel services such as the error detection and reporting facility and the core dump facility. For information about configuration, see *Persistent Memory Configuration*, p.250.

Reserved Memory Configuration: User-Reserved Memory and Persistent Memory

Two types of reserved memory can be configured in VxWorks: user-reserved memory and persistent memory. Information about the reserved memory can be obtained with **userReservedGet( )**.

For more information, see *Persistent Memory*, p.250 and *User-Reserved Memory*, p.250, as well as Figure 10-2.

User-Reserved Memory Configuration
: User-reserved memory is enabled with the **INCLUDE_USER_RESERVED_MEMORY** component. Additionally, the BSP must reserved memory entry via the **sysMemDescGet( )** function. The **INCLUDE_USER_RESERVED_MEMORY** component has the following parameters:

USER_RESERVED_MEM
: Defines the size of the user-reserved memory region. The default value is 0.

CLEAR_USER_RESERVED_MEMORY_ON_COLD_BOOT,
: Clears user-reserved memory when set to **TRUE** (the default).

Persistent Memory Configuration
: Persistent memory is enabled with the **INCLUDE_EDR_PM** component and configured with the **PM_RESERVED_MEM** parameter.

System RAM Autosizing

For information about system RAM autosizing, see the relevant BSP reference.

Kernel Common Heap

The kernel common heap is the memory partition used by the kernel and kernel applications for dynamic memory allocation. It is managed using the standard ANSI memory allocation functions, **malloc( )**, **free( )**, and so on. For more information see also the **memPartLib** and **memLib** API references.

The kernel common heap is also used for allocating memory for kernel modules that are downloaded from the host system. By default, memory for downloaded kernel modules is allocated from the kernel proximity heap (see *Kernel Proximity Heap*, p.251).

The initial size of the heap is defined by the **KERNEL_COMMON_HEAP_INIT_SIZE** kernel configuration parameter (size set in bytes). The kernel common heap supports auto-growth. When the current heap content cannot satisfy the requested allocation the operating system will automatically add more memory that is allocated from the kernel virtual memory pool region and the global RAM pool (subject to available free RAM in the pool). The amount of the heap growth is a multiple of the **KERNEL_COMMON_HEAP_INCR_SIZE** kernel configuration parameter (size set in bytes). Setting the parameter to zero disables auto-growth.

There is no limit for the number of segments in the partition. As the heap guarantees that every block allocated is contiguous in physical address space, there is a system specific limit of the memory partition segments based on the availability of physically contiguous memory in the global RAM pool. The segment size determines the largest block that can be allocated from the partition. Note that **mmap( )** based memory allocation is not subject to the physically contiguous requirement.

The **KERNEL_COMMON_HEAP_INIT_SIZE** parameter should not be set to a value larger than the available largest physically contiguous block of RAM. Take into account the size of a page and other bookkeeping information, which reduces the actual maximum value that can be used with this parameter.

Kernel Proximity Heap

The kernel proximity heap is a memory partition created in the kernel system region and is managed using the **kProxHeapLib** API.

The kernel proximity heap is also used (by default) for allocating memory for kernel modules that are downloaded from the host system. The modules must be compiled for the kernel code model, which is the default. Memory can also be allocated for modules from the kernel common heap (see *Kernel Common Heap*, p.250).

In addition, the kernel proximity heap is used to allocate memory for interrupt stubs, or other blocks of memory that contain executable code.

The kernel proximity heap is the first allocation facility initialized. During the early phase of the system startup (before the kernel common heap is created) VxWorks libraries use it to perform general-purpose allocations that are not necessarily subject to code model requirements.

The kernel proximity is created from the free portion of the kernel system RAM (that is, not used for kernel code and so on). The size of the kernel proximity heap is set with the **KERNEL_PROXIMITY_HEAP_SIZE** parameter.

DMA32 Heap

The DMA32 heap is a memory partition used by drivers for devices that do not have the capability to address more than 4 GB of physical address space (such as 32-bit PCI devices). The DMA32 heap is managed using the **cacheDma32Lib** API.

Typically, the physical memory for this heap is located in RAM in the lower 4 GB, as provided by the BSP configuration. Virtual memory is allocated from the kernel virtual memory pool region.

10.3 **Physical Memory Mapping Library**

VxWorks provides the **pmapBaseLib** library to map or unmap physical addresses for the kernel and RTP context.

This allows drivers and other modules such as the graphics layer to use a standard interface to map and unmap physical addresses. To use the physical address mapping library in your VxWorks image, add the **INCLUDE_PMAP_LIB** component to your VIP. For more information, see the API reference for **pmapBaseLib**.

10.4 **VxWorks Memory Allocation Facilities**

Table 10-1 **Memory Allocation Facilities**

Facility	**Key Characteristics**	**Advantages**
Memory partitions	Most general purpose. The foundation for the kernel and RTP heap.	ANSI standard allocation functions: **malloc( )**, **free( )**, and so on. Most flexible (that is, any size, any alignment). Additional partitions can be created. For more information, see *10.6 VxWorks Kernel Heap and Memory Partition Management*, p.254.
Memory pools	Used for allocation of fixed size items. Multiple pools can be created for differently sized objects.	Very fast. Not susceptible to fragmentation. Option for internal thread-safety (can be disabled if user guaranties thread safety at higher level). Option for specific alignment. For more information, see *10.8 Memory Pools*, p.256.
Private **mmap( )**	MMU page granularity (usually 4 K) and alignment.	Best for allocating large blocks of memory. **munmap( )** frees memory to global RAM pool immediately—as opposed to heap growth, which is unidirectional. Memory added to the heap remains there permanently for the kernel heap, or until the RTP gets deleted. For more information see *10.10 Memory Mapping Facilities*, p.259.

10.5 RTP Heap and Memory Partition Management

VxWorks provides support for heap and memory partition management in real-time processes (RTPs). 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 increment 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 *RTPs and Environment Variables*, p.19, *3.2 RTP Application Structure*, p.26, 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. This solution is simple, but it creates a fixed sized heap. For example:

  ```
  char heapMem[HEAP_SIZE];
  ```

- 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 *POSIX Memory Management APIs*, p.257.

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**.

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 function is called.

- Have its initialization function 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 *Library and Plug-in Initialization*, p.46).

10.6 VxWorks Kernel Heap and Memory Partition Management

The kernel common heap is the memory partition used by the kernel and kernel applications for dynamic memory allocation. The kernel proximity heap is a memory partition created in the kernel system region. Its location guarantees that it satisfies the linkage and relocation requirements for the *kernel code model* used to build the kernel image. The DMA32 heap is a memory partition used by drivers for devices that do not have the capability to address more than 4 GB of physical address space.

Kernel Common Heap

The kernel common heap is the memory partition used by the kernel and kernel applications for dynamic memory allocation. It is managed using the standard ANSI memory allocation functions, **malloc( )**, **free( )**, and so on. For more information see also the **memPartLib** and **memLib** API references.

The kernel common heap is also used for allocating memory for kernel modules that are downloaded from the host system. The modules must be compiled for the *large code model*, and the **LOAD_COMMON_HEAP** option must used with the loader APIs. By default, memory for downloaded kernel modules is allocated from the kernel proximity heap (see *Kernel Proximity Heap*, p.251). For information about the kernel object-module loader, see the *VxWorks Kernel Shell User's Guide*.

The initial size of the heap is defined by the **KERNEL_COMMON_HEAP_INIT_SIZE** kernel configuration parameter (size set in bytes). The kernel common heap supports auto-growth. When the current heap content cannot satisfy the requested allocation the operating system will automatically add more memory that is allocated from the kernel virtual memory pool region and the global RAM pool (subject to available free RAM in the pool). The amount of the heap growth is a multiple of the **KERNEL_COMMON_HEAP_INCR_SIZE** kernel configuration parameter (size set in bytes). Setting the parameter to zero disables auto-growth.

Each segment added to the kernel common heap memory partition is subject to an architectural limit of 16 GB. There is no limit for the number of segments in the partition. As the heap guarantees that every block allocated is contiguous in physical address space, there is a system specific limit of the memory partition segments based on the availability of physically contiguous memory in the global RAM pool. The segment size determines the largest block that can be allocated from the partition. Note that **mmap( )** based memory allocation is not subject to the physically contiguous requirement.

The segment size limit of 16 GB applies to the initial heap block as well. Therefore the **KERNEL_COMMON_HEAP_INIT_SIZE** parameter should not be set to a value larger than the available largest physically contiguous block of RAM. Take into account the size of a page and other bookkeeping information, which reduces the actual maximum value that can be used with this parameter.

Kernel Proximity Heap

The kernel proximity heap is a memory partition created in the kernel system region. The term *proximity* refers to the location close to the kernel code. The location guarantees that it satisfies the linkage and relocation requirements for the *kernel code model* used to build the kernel image (such as the kernel code model defined by the ABI for the x86-64 architecture). The kernel proximity heap is managed using the **kProxHeapLib** API.

The kernel proximity heap is also used (by default) for allocating memory for kernel modules that are downloaded from the host system. The modules must be compiled for the kernel code model, which is the default. Memory can also be allocated for modules from the kernel common heap (see *Kernel Common Heap*, p.250). For information about the kernel object-module loader, see the *VxWorks Kernel Shell User's Guide*.

In addition, the kernel proximity heap is used to allocate memory for interrupt stubs, or other blocks of memory that contain executable code.

The kernel proximity heap is the first allocation facility initialized. During the early phase of the system startup (before the kernel common heap is created) VxWorks libraries use it to perform general-purpose allocations that are not necessarily subject to code model requirements.

The kernel proximity is created from the free portion of the kernel system RAM (that is, not used for kernel code, WDB memory, and so on). The size of the kernel proximity heap is set with the **KERNEL_PROXIMITY_HEAP_SIZE** parameter.

DMA32 Heap

The DMA32 heap is a memory partition used by drivers for devices that do not have the capability to address more than 4 GB of physical address space (such as 32-bit PCI devices). The DMA32 heap is managed using the **cacheDma32Lib** API.

Typically, the physical memory for this heap is located in RAM in the lower 4 GB, as provided by the BSP configuration. Virtual memory is allocated from the kernel virtual memory pool region.

10.7 Optimized Kernel Memory Allocation

The **memPartCacheLib** library provides an extension to the VxWorks kernel heap manager, **memPartLib**, to improve execution speed of tasks that frequently perform dynamic memory allocation.

The speedup is achieved by reducing the contention to the critical sections of the heap manager, which are serialized by a mutex semaphore. The implementation relies on task-private data structures so that a task can usually allocate and free memory blocks without having to use locks. Conceptually, this acts as a cache of memory blocks that can be quickly reused when needed. The memory assigned to each task is divided into various bins where each bin represents allocation of a given size. The **memPartCacheLib** services allocation and free requests of size up to 512 bytes via the **malloc( )** and **free( )** functions. Allocations of size greater than 512 bytes are passed back to the standard heap allocator. The bin sizes are spaced so that the small sizes are separated by 16 bytes each, larger sizes are separated by 32, and so on. The maximum spacing is 64 bytes.

This feature is available for both SMP and uniprocessor configurations. For SMP systems it can help to reduce heap contentions between tasks running on multiple cores. For uniprocessor systems it can provide similar performance improvements. In either case, the amount of improvement depends on the allocation pattern of the application.

For additional information, see the API reference for **memPartCacheLib**.

Configuration

To enable task level memory caching of the system heap, configure VxWorks with the **INCLUDE_MEM_PART_CACHE** component.

By default **MEM_PART_CACHE_GLOBAL_ENABLE** configuration parameter is set to **FALSE**, which provides for task-level memory caching for individual tasks. However, it is not enabled until **memPartCacheCreate( )** is called in the context of a given task.

If the parameter is set to **TRUE**, the task level memory caching is enabled globally for all tasks. When any task is created, the task level memory caching is enabled automatically. When the task is deleted, any memory that is cached in the memory cache is released.

10.8 Memory Pools

A memory pool is a dynamic set of statically sized memory items. Pools provide a fast and efficient memory management for applications that use a large number of identically sized memory items (structures, objects, and so on) by minimizing the number of allocations from a memory partition. The use of memory pools also reduces possible fragmentation caused by frequent memory allocation and freeing.

Memory pools are designed for use in systems requiring frequent allocating and freeing of memory in statically sized blocks, such as used in messaging systems, data- bases, and so on. The pool system is dynamic and if necessary, can automatically grow by allocating a user-specified number of new items from a memory partition or from the heap.

For more information, see the **poolLib** API reference entry.

10.9 POSIX Memory Management

VxWorks provides various POSIX memory management features.

The features are as follows:

- Dynamic memory allocation—with the **calloc( ) malloc( ) realloc( )** and **free( )** functions. See*10.5 RTP Heap and Memory Partition Management*, p.253 and *10.6 VxWorks Kernel Heap and Memory Partition Management*, p.254.
- POSIX memory-mapped files (the **_POSIX_MAPPED_FILES** option). See *POSIX Memory-Mapped Files*, p.262.
- POSIX shared memory objects (the **_POSIX_SHARED_MEMORY_OBJECTS** option). See *POSIX Shared Memory Objects*, p.262.

- POSIX memory protection (the **_POSIX_MEMORY_PROTECTION** options). See *POSIX Memory Protection*, p.258.
- POSIX memory locking (the **_POSIX_MEMLOCK** and **_POSIX_MEMLOCK_RANGE** options). See *POSIX Memory Locking*, p.258.

In addition, VxWorks also supports anonymous memory mapping, which is not a POSIX standard, but an implementation-specific extension of it.

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** functions are listed in Table 10-2.

Table 10-2 **POSIX Memory Management Functions**

Function	Description
mmap( )	Establishes a mapping between an RTP or kernel 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 function does nothing.
munlockall( )	Unlocks all pages used by a process. In VxWorks this function does nothing.
mlock( )	Locks specified pages into memory. In VxWorks this function does nothing.
munlock( )	Unlocks specified pages. In VxWorks this function does nothing.
shm_open( )	Opens a shared memory object.
shm_unlink( )	Removes a shared memory object.

Restrictions on mmanLib Implementation

The restrictions on the VxWorks implementation of **mmanLib** are as follows:

- Mapping at fixed a address (**MAP_FIXED**) is not supported.
- The **munmap( )**, **mprotect( )**, and **msync( )** functions only succeed for memory pages obtained with **mmap( )**.
- The **msync( )** function must be called explicitly to ensure that data for memory mapped files is synchronized with the media.
- VxWorks provides the POSIX memory locking functions: **mlock( )**, **munlock( )**, **mlockall( )**, and **munlockall( )**. However, memory mappings in VxWorks are always memory-resident. This ensures deterministic memory access for mapped files, but it also means that physical memory is continuously associated with mappings, until it is unmapped. Therefore, these POSIX memory locking functions do not do anything, and are provided simply for application portability.
- 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.

POSIX Memory Mapping

The POSIX memory mapping features are discussed in *10.10 Memory Mapping Facilities*, p.259.

POSIX Memory Protection

Memory protection of mappings established with **mmap( )** can be changed using the **mprotect( )** function. It works with both anonymous memory mapped files and with shared memory objects.

For more information see the **mprotect( )** API reference.

POSIX 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.

10.10 Memory Mapping Facilities

VxWorks provides memory mapping facilities for memory mapped files, shared memory objects, anonymous memory mapping, device memory, and shared data regions.

The facilities are provided as follows:

- Memory mapped files—a POSIX facility using **mmanLib**.
- Shared memory objects—a POSIX facility using **mmanLib** and **shmLib**.
- Anonymous memory mapping—a VxWorks extension to POSIX memory mapping using **mmanLib** with an anonymous flag.
- Device memory—a VxWorks extension to POSIX memory mapping using **mmanLib** and **devMemLib**.
- Shared data regions—a VxWorks facility using **sdLib**.

The memory mapping features described in this section build on the functionality provided by the lower-level virtual memory management facility and the address space management facility.

For more information about address space management see the **adrSpaceLib** API reference entry.

Table 10-3 provides a comparison of the POSIX-based (**mmanLib**) features and extensions, and proprietary (**sdLib**) memory features, which overlap in functionality.

NOTE: Wind River recommends using **mmanLib** (and associated features **shmLib** and **devMemLib**) instead of **sdLib** (shared data regions) because of the additional functionality and POSIX compliance.

Table 10-3 **Memory-Mapped File Facilities**

Feature	mmanLib	sdLib	Comments
standard API	yes	no	**shmLib** and **mmanLib** are POSIX. **sdLib** is proprietary.
supported in RTP	yes	yes	
supported in kernel	yes	yes	
private mapping	yes	yes	**sdlib** private mapping prohibits any other RTP from re-mapping shared data regions (the creating RTP has exclusive access to it). **mmanLib** conforms to the POSIX standard, allowing multiple private mappings of a shared memory object or regular file.
unnamed (anonymous)	yes	no	With **mmap( )** no object needs to be created or opened; it can be easily used to allocate additional memory to either the kernel or RTP VM context. After use the memory can be freed to the global pool with **munmap( )**.
shared memory	yes	yes	**mmap( )** supports POSIX shared memory objects (PSE52).

Table 10-3 **Memory-Mapped File Facilities** (cont'd)

Feature	mmanLib	sdLib	Comments
regular files	yes	no	**mmap( )** can be used to map regular files as long as the file system provides POSIX-conforming inode information (for example, HRFS).
device mapping	yes	no	**mmap( )** device mapping is implemented as an extension to the POSIX API (but similar to POSIX shared memory objects). Device mapping provides the ability to map to a fixed physical address.
global mapping	yes	no	Global mappings refer to mappings that can be accessed from any virtual memory context—the kernel and all RTPs (during system calls). It can also be safely accessed in ISR context. This includes most kernel mappings: text, data, bss, kernel heap, and anything mapped with **mmap( )** by a kernel task. All mappings created with **mmap( )** in the kernel are global. It does not include shared data regions mapped by a kernel task. For shared data regions in the kernel, a significant limitation is that they cannot be safely accessed from interrupt service functions because they are always mapped private to the specific context, not global.
kernel protection	yes	yes	Kernel memory should not be re-mapped by an RTP when specifying a physical address to be mapped. Shared data relies on an inverse page table, which can add significant runtime footprint and prevents remapping of any physical address. With **mmap( )**, creation of device memory objects is allowed by kernel tasks only. RTPs can open device memory objects that have already been created in the kernel; a physical address can be alias-mapped, if necessary.
common virtual address	yes	yes	Shared data regions are all mapped using the same virtual address in all RTPs and the kernel. With **mmap( )**, shared mappings use a common address for all RTPs, but a different address for the kernel. Note that applications should not in any case share pointers between RTPs. In other words, applications should not expect the same virtual address to be assigned (most other operating systems allocate different virtual address in different process).
name space	file system	named object	
identifier	file descriptor	object ID	

Table 10-3 **Memory-Mapped File Facilities** (cont'd)

Feature	mmanLib	sdLib	Comments
regular files	yes	no	**mmap( )** can be used to map regular files as long as the file system provides POSIX-conforming inode information (for example, HRFS).
device mapping	yes	no	**mmap( )** device mapping is implemented as an extension to the POSIX API (but similar to POSIX shared memory objects). Device mapping provides the ability to map to a fixed physical address.
global mapping	yes	no	Global mappings refer to mappings that can be accessed from any virtual memory context—the kernel and all RTPs (during system calls). It can also be safely accessed in ISR context. This includes most kernel mappings: text, data, bss, kernel heap, and anything mapped with **mmap( )** by a kernel task. All mappings created with **mmap( )** in the kernel are global. It does not include shared data regions mapped by a kernel task. For shared data regions in the kernel, a significant limitation is that they cannot be safely accessed from interrupt service functions because they are always mapped private to the specific context, not global.
kernel protection	yes	yes	Kernel memory should not be re-mapped by an RTP when specifying a physical address to be mapped. Shared data relies on an inverse page table, which can add significant runtime footprint and prevents remapping of any physical address. With **mmap( )**, creation of device memory objects is allowed by kernel tasks only. RTPs can open device memory objects that have already been created in the kernel; a physical address can be alias-mapped, if necessary.
common virtual address	yes	yes	Shared data regions are all mapped using the same virtual address in all RTPs and the kernel. With **mmap( )**, shared mappings use a common address for all RTPs, but a different address for the kernel. Note that applications should not in any case share pointers between RTPs. In other words, applications should not expect the same virtual address to be assigned (most other operating systems allocate different virtual address in different process).
name space	file system	named object	
identifier	file descriptor	object ID	

Table 10-3 **Memory-Mapped File Facilities** (cont'd)

Feature	mmanLib	sdLib	Comments
ability to map at fixed virtual address	no	no	
unit of operation	MMU page	entire shared data region	*Unit of operation* means the portion of shared memory or a shared data region that can be operated on (mapped, unmapped, or protected). For **mmanLib** 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.

POSIX Memory-Mapped Files

VxWorks provides support for POSIX memory-mapped files. For memory-mapped files, the file descriptor that is used when calling **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 **INCLUDE_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 the 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.

POSIX Shared Memory Objects

VxWorks provides support for POSIX shared memory objects. With this type of mapping, the file descriptor that is used when calling **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 **INCLUDE_POSIX_MAPPED_FILES** components, respectively.

The shared memory file system provides the name space for shared memory objects. It is a virtual 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.

For more information, see the **shmLib** and **mmanLib** API references. For a code example, see the **shmLib** API.

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. Only private mappings are supported with the anonymous option.

Anonymous memory-mapping is automatically included for user-mode applications when basic process (RTP) support included in VxWorks. The **INCLUDE_MMAP** component must be added to provide support for support in the kernel.

The following RTP application example shows the use of **mmap( )** with the anonymous flag (**MAP_ANONYMOUS**), as well as **mprotect( )**, and **unmap( )**. A kernel application would essentially use the same code, with the exception of the **main( )** function having a different name.

```
#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 ()
    {
    size_t pgSize;     /*variable to store page size */
    size_t bufSize;            /* size of buffer */
    char * pBuf;            /* buffer  */

    pgSize=vmPageSizeGet();

    /* 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_ANONYMOUS), 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** functions, and an additional code example, see the VxWorks API reference for **mmanLib**.

Device Memory Objects

VxWorks provides a non-POSIX extension to the memory-mapped files facility that allows for accessing device memory, which can be used for any device driver (serial, network, graphic, and so on). The facility is provided by the **INCLUDE_DEVMEM** component.

The **devMemCreate( )**, **devMemOpen( )**, and **devMemUnlink( )** kernel functions create, open, and remove a device memory object. The **mmap( )** function is used to map the device memory object after it has been created and opened. With this kind of mapping, the file descriptor that is used when calling **mmap( )** is obtained with **devMemOpen( )**.

For more information, see API reference for **devMemLib**.

Shared Data Regions

The proprietary shared data regions facility provides a means for RTP applications to share a common area of memory with each other. RTPs otherwise provide for full separation and protection of all processes from one another.

NOTE: Wind River recommends using **mmanLib** (and associated features **shmLib** and **devMemLib**) instead of **sdLib** (shared data regions) because of the additional functionality and POSIX compliance.

10.11 Kernel Virtual Memory Management

VxWorks can be configured with an architecture-independent interface to the CPU's memory management unit (MMU) to provide virtual memory support.

This support includes the following features:

- Setting up the kernel memory context at boot time.
- Mapping pages in virtual space to physical memory.
- Setting caching attributes on a per-page basis.
- Setting protection attributes on a per-page basis.
- Setting a page mapping as valid or invalid.
- Locking and unlocking TLB entries for pages of memory.
- Enabling page optimization.

The programmable elements of virtual memory (VM) support are provided by the **vmBaseLib** library.

NOTE: There are differences in the **vmBaseLib** library provided for the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks, and special guidelines for its use in optimizing SMP applications. For more information about **vmBaseLib** and SMP, see *vmBaseLib Restrictions*, p.454 and *Using vmBaseLib*, p.446. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

When RTP support is included in VxWorks with the **INCLUDE_RTP** component, the virtual memory facilities also provide system support for managing multiple virtual memory contexts, such as creation and deletion of RTP memory context.

For information about additional MMU-based memory protection features beyond basic virtual memory support, see *10.12 Additional Kernel Memory Protection Features*, p.272.

Also note that errors (exceptions) generated with the use of virtual memory features can be detected and managed with additional VxWorks facilities. See *13. Error Detection and Reporting* for more information.

Configuring Virtual Memory Management

The components listed in Table 10-4 provide basic virtual memory management, as well as show functions for use from the shell.

Table 10-4 **MMU Components**

Constant	Description
INCLUDE_MMU_GLOBAL_MAP	Initialize the kernel's global MMU mappings according to the BSP's **sysPhysMemDesc[ ]** table. See *Configuring the Kernel Virtual Memory Context*, p.265.
INCLUDE_MMU_BASIC	Include the **vmBaseLib** API, which is used for programmatic management of virtual memory (see *Managing Virtual Memory Programmatically*, p.267).
INCLUDE_LOCK_TEXT_SECTION	Kernel text TLB locking optimization.
INCLUDE_PAGE_SIZE_OPTIMIZATION	Page size optimization for the kernel.
INCLUDE_VM_SHOW	Virtual memory show functions for the shell C interpreter.
INCLUDE_VM_SHOW_SHELL_CMD	Virtual memory show commands for the shell command interpreter.

For information about related components see *10.12 Additional Kernel Memory Protection Features*, p.272.

Configuring the Kernel Virtual Memory Context

The kernel virtual memory context is created automatically at boot time based on configuration data provided by the BSP. The primary data is in the

sysPhysMemDesc[ ] table, which is usually defined in the BSP's **sysLib.c** file. The table defines the initial kernel mappings and initial attributes. The entries in this table are of **PHYS_MEM_DESC** structure type, which is defined in **vmLib.h**.

There is usually no need to change the default **sysPhysMemDesc[ ]** configuration. However, modification may be required or advisable, for example, when:

- New driver support or new devices (for example, flash memory) are added to the system.
- The protection or cache attributes of certain entries must be changed. For example, entries for flash memory can be read-only if the content of the flash is never written from VxWorks. However, if a flash driver such as TrueFFS is used, the protection attribute has to be set to writable.
- There are unused entries in the table. In general, it is best to keep only those entries that actually describe the system, as each entry may require additional system RAM for page tables (depending on size of the entry, its location relative to other entries, and architecture-specific MMU parameters). The larger the memory blocks mapped, the more memory is used for page tables.

The **sysPhysMemDesc[ ]** table can be modified at run-time. This is useful, for example, with PCI drivers that can be auto-configured, which means that memory requirements are detected at run-time. In this case the size and address fields can be updated programmatically for the corresponding **sysPhysMemDesc[ ]** entries. It is important to make such updates before the VM subsystem is initialized by **usrMmuInit( )**, for example during execution of **sysHwInit( )**.

For more information, see the *VxWorks 7 BSP Developer's Guide*.

CAUTION: The regions of memory defined in **sysPhysMemDesc[ ]** must be page-aligned, and must span complete pages. In other words, the first three fields (virtual address, physical address, and length) of a **PHYS_MEM_DESC** structure must all be even multiples of the MMU page size. Specifying elements of **sysPhysMemDesc[ ]** that are not page-aligned causes the target to reboot during initialization. See the *VxWorks 7 Architecture Supplement* to determine what page size is supported for the architecture in question.

Configuration Example

This example is based on multiple CPUs using the shared-memory network. A separate memory board is used for the shared-memory pool. Because this memory is not mapped by default, it must be added to **sysPhysMemDesc[ ]** for all the boards on the network. The memory starts at 0x4000000 and must be made non-cacheable, as shown in the following code fragment:

```
/* shared memory */
{
(VIRT_ADDR)  0x4000000,          /* virtual address */
(PHYS_ADDR)  0x4000000,          /* physical address */
0x20000,                         /* length */
/* initial state mask */
MMU_ATTR_VALID_MSK | MMU_ATTR_PROT_MSK | MMU_ATTR_CACHE_MSK,
/* initial state */
MMU_ATTR_VALID | MMU_ATTR_PROT_SUP_READ | MMU_ATTR_PROT_SUP_WRITE |
MMU_ATTR_CACHE_OFF
}
```

For some architectures, the system RAM (the memory used for the VxWorks kernel image, kernel heap, and so on) must be identity mapped. This means that for the corresponding entry in the **sysPhysMemDesc[ ]** table, the virtual address must be the same as the physical address. For more information see *10.1 VxWorks 6.9-Compatible Memory Model for 32-Bit BSPs*, p.245 and the *VxWorks Architecture Supplement*.

Managing Virtual Memory Programmatically

This section describes the facilities provided for manipulating the MMU programmatically using low-level functions in **vmBaseLib**. You can make portions of memory non-cacheable, write-protect portions of memory, invalidate pages, lock TLB entries, or optimize the size of memory pages.

For more information about the virtual memory functions, see the VxWorks API reference for **vmBaseLib**.

NOTE: There are differences in the **vmBaseLib** library provided for the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks, and special guidelines for its use in optimizing SMP applications. For more information about **vmBaseLib** and SMP, see *vmBaseLib Restrictions*, p.454 and *Using vmBaseLib*, p.446. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

Modifying Page States

Each virtual memory page (typically 4 KB) has a state associated with it. A page can be valid/invalid, readable, writable, executable, or cacheable/non-cacheable.

The state of a page can be changed with the **vmStateSet( )** function. See Table 10-5 and Table 10-6 for lists of the page state constants and page state masks that can be used with **vmStateSet( )**. A page state mask must be used to describe which flags are being changed. A logical OR operator can be used with states and masks to define both mapping protection and cache attributes.

Table 10-5 **Page State Constants**

Constant	Description
MMU_ATTR_VALID	Valid translation
MMU_ATTR_VALID_NOT	Invalid translation
MMU_ATTR_PRO_SUP_READ	Readable memory in kernel mode
MMU_ATTR_PRO_SUP_WRITE	Writable memory in kernel mode
MMU_ATTR_PRO_SUP_EXE	Executable memory in kernel mode
MMU_ATTR_PRO_USR_READ	Readable memory in user mode
MMU_ATTR_PRO_USR_WRITE	Writable memory in user mode
MMU_ATTR_PRO_USR_EXE	Executable memory in user mode
MMU_ATTR_CACHE_OFF	Non-cacheable memory
MMU_ATTR_CACHE_COPYBACK	Cacheable memory, copyback mode
MMU_ATTR_CACHE_WRITETHRU	Cacheable memory, writethrough mode

Table 10-5 **Page State Constants** (cont'd)

Constant	Description
MMU_ATTR_CACHE_DEFAULT	Default cache mode (equal to either **COPYBACK**, **WRITETHRU**, or **CACHE_OFF**, depending on the setting of **USER_D_CACHE_MODE**)
MMU_ATTR_CACHE_COHERENCY	Memory coherency is enforced (not supported on all architectures; for more information, see the *VxWorks7 Architecture Supplement*)
MMU_ATTR_CACHE_GUARDED	Prevent out-of-order load operations, and pre-fetches (not supported on all architectures; for more information, see the *VxWorks 7 Architecture Supplement*)
MMU_ATTR_NO_BLOCK	Page attributes can be changed from ISR.
MMU_ATTR_SPL_0 ... **MMU_ATTR_SPL_7**	Optional Architecture Specific States (only used by some architectures; for more information, see the *VxWorks 7 Architecture Supplement*)

Table 10-6 **Page State Masks**

Constant	Description
MMU_ATTR_VALID_MSK	Modify valid flag
MMU_ATTR_PROT_MSK	Modify protection flags
MMU_ATTR_CACHE_MSK	Modify cache flags
MMU_ATTR_SPL_MSK	Modify architecture specific flags

Not all combinations of protection settings are supported by all CPUs. For example, many processor types do not provide setting for execute or non-execute settings. On such processors, readable also means executable.

For information about architecture-specific page states and their combination, see the *VxWorks 7 Architecture Supplement*.

Making Memory Non-Writable

Sections of memory can be write-protected using **vmStateSet( )** to prevent inadvertent access. This can be used, for example, to restrict modification of a data object to a particular function. If a data object is global but read-only, tasks can read the object but not modify it. Any task that must modify this object must call the associated function. Inside the function, the data is made writable for the duration of the function, and on exit, the memory is set to **MMU_ATTR_PROT_SUP_READ**.

Nonwritable Memory Example

In this code example, a task calls **dataModify( )** to modify the data structure pointed to by **pData**. This function makes the memory writable, modifies the data, and sets the memory back to nonwritable. If a task subsequently tries to modify the data without using **dataModify( )**, a data access exception occurs.

```
/* privateCode.h - header file to make data writable from function only */
#define MAX 1024
typedef struct myData
    {
    char stuff[MAX];
    int moreStuff;
    } MY_DATA;

/* privateCode.c - uses VM contexts to make data private to a code segment */
#include <vxWorks.h>
#include <string.h>
#include <vmLib.h>
#include <semLib.h>
#include "privateCode.h"
MY_DATA * pData;
SEM_ID dataSemId;
int pageSize;
/***********************************************************************
*
* initData - allocate memory and make it nonwritable
*
* This function initializes data and should be called only once.
*
*/
STATUS initData (void)
    {
    pageSize = vmPageSizeGet();
    /* create semaphore to protect data */
    dataSemId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
    /* allocate memory = to a page */
    pData = (MY_DATA *) valloc (pageSize);
    /* initialize data and make it read-only */
    bzero ((char *) pData, pageSize);
    if (vmStateSet (NULL, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    /* release semaphore */
    semGive (dataSemId);
    return (OK);
    }
/********************************************************************
*
* dataModify - modify data
*
* To modify data, tasks must call this function, passing a pointer to
* the new data.
* To test from the shell use:
*     -> initData
*     -> sp dataModify
*     -> d pData
*     -> bfill (pdata, 1024, 'X')
*/
STATUS dataModify
    (
    MY_DATA * pNewData
    )
    {
    /* take semaphore for exclusive access to data */
    semTake (dataSemId, WAIT_FOREVER);
    /* make memory writable */
    if (vmStateSet (NULL, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ | MMU_ATTR_PROT_SUP_WRITE) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    /* update data*/
    bcopy ((char *) pNewData, (char *) pData, sizeof(MY_DATA));
    /* make memory not writable */
    if (vmStateSet (NULL, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    semGive (dataSemId);
    return (OK);
    }
```

Invalidating Memory Pages

To invalidate memory on a page basis, use **vmStateSet( )** as follows:

```
vmStateSet (NULL, address, len, MMU_ATTR_VALID_MSK, MMU_ATTR_VALID_NOT);
```

Any access to a mapping made invalid generates an exception whether it is a read or a write access.

To re-validate the page, use **vmStateSet( )** as follows:

```
vmStateSet (NULL, address, len, MMU_ATTR_VALID_MSK, MMU_ATTR_VALID);
```

Locking TLB Entries

For some processors it is possible to force individual entries in the Translation Look-aside Buffer (TLB) to remain permanently in the TLB. When the architecture-specific MMU library supports this feature, the **vmPageLock( )** function can be used to lock page entries, and **vmPageUnlock( )** to unlock page entries.

The **INCLUDE_LOCK_TEXT_SECTION** component provides facilities for TLB locking. When this component is included in VxWorks, the kernel image text section is automatically locked at system startup.

This feature can be used for performance optimizations in a manner similar to cache locking. When often-used page entries are locked in the TLB, the number of TLB misses can be reduced. Note that the number of TLB entries are generally limited on all processors types, so locking too many entries can result in contention for the remaining entries that are used dynamically.

For more information, see the *VxWorks Architecture Supplement*.

Page Size Optimization

For some processors it is possible to enable larger page sizes than the default (defined by **VM_PAGE_SIZE**) for large, contiguous memory blocks that have homogeneous memory attributes and satisfy the physical and virtual address alignment requirements. There are several advantages to using such optimization, including:

- Reducing the number of page table entries (PTE) needed to map memory, resulting in less memory used.
- More efficient TLB entry usage, resulting in fewer TLB misses, therefore potentially better performance.

For 32-bit VxWorks, optimization of the entire kernel memory space (including I/O blocks) at startup can be accomplished by configuring VxWorks with the **INCLUDE_PAGE_SIZE_OPTIMIZATION** component. Page size optimization for specific blocks of memory can be accomplished at run-time with the **vmPageOptimize( )** function.

For 64-bit VxWorks, all mappings (kernel and RTP context) are performed with page optimization enabled when the **INCLUDE_PAGE_SIZE_OPTIMIZATION** component is included. It is included by default for 64-bit VxWorks.

De-optimization is performed automatically when necessary. For example, if part of a memory block that has been optimized is set with different attributes, the large page is automatically broken up into multiple smaller pages and the new attribute is set to the requested pages only.

Setting Page States in ISRs

For many types of processors, **vmStateSet( )** is a non-blocking function, and can therefore be called safely from ISRs. However, it may block in some cases, such as on processors that support page size optimization (see *Page Size Optimization*, p. 270).

To make sure that **vmStateSet( )** can be called safely from an ISR for specific pages, the page must first have the **MMU_ATTR_NO_BLOCK** attribute set. The following code example shows how this can be done:

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

 #define DATA_SIZE   0x10000

 char * pData;

 void someInitFunction ()
     {
     /* allocate buffer */

     pData = (char *) valloc (DATA_SIZE);

     /* set no-block attribute for the buffer */

     vmStateSet (NULL, (VIRT_ADDR) pData, DATA_SIZE,
                 MMU_ATTR_SPL_MSK, MMU_ATTR_NO_BLOCK);
     }

 void someISR ()
     {
     ...
     /* now it's safe to set any attribute for the buffer in an ISR */

     vmStateSet (NULL, (VIRT_ADDR) pData, DATA_SIZE,
                 MMU_ATTR_PROT_MSK, MMU_ATTR_SUP_RWX);
     ...
     }
```

Troubleshooting

The show functions and commands described in Table 10-7 are available to assist with trouble-shooting virtual memory problems.

Table 10-7 **Virtual Memory Shell Commands**

C Interpreter	Command Interpreter	Description
vmContextShow( )	**vm context**	Lists information about the entire RTP context, including private mappings and kernel mappings (for supervisor access), as well as any shared data contexts attached to the RTP.
rtpMemShow( )	**rtp meminfo**	Lists only the RTP's private mappings.

These functions and commands are provided by the **INCLUDE_VM_SHOW**, **INCLUDE_VM_SHOW_SHELL_CMD**, **INCLUDE_RTP_SHOW**, and **INCLUDE_RTP_SHOW_SHELL_CMD** components.

For more details and usage example of the show functions see the VxWorks shell references.

10.12 Additional Kernel Memory Protection Features

VxWorks provides MMU-based features for the kernel that supplement basic virtual memory support to provide a more reliable run-time environment.

These additional memory-protection features are as follows:

- task stack overrun and underrun detection
- interrupt stack overrun and underrun detection
- non-executable task stacks
- text segment write-protection
- exception vector table write-protection

For information about basic virtual memory support, see *10.11 Kernel Virtual Memory Management*, p.264.

Errors generated with the use of these features can be detected and managed with additional VxWorks facilities. See *13. Error Detection and Reporting* for more information.

Configuring VxWorks for Additional Memory Protection

The components listed in Table 10-8 provide additional memory-protection features. They can be added to VxWorks as a unit with the **INCLUDE_KERNEL_HARDENING** component. The individual and composite components all include the basic virtual memory component **INCLUDE_MMU_BASIC** by default.

Table 10-8 **Additional Memory Protection Components**

Component	Description
INCLUDE_PROTECT_TASK_STACK	Task stack overrun and underrun protection.
INCLUDE_TASK_STACK_NO_EXEC	Non-executable task stacks.
INCLUDE_PROTECT_TEXT	Text segment write-protection.
INCLUDE_PROTECT_VEC_TABLE	Exception vector table write-protection and NULL pointer reference detection.
INCLUDE_PROTECT_INTERRUPT_STACK	Interrupt stack overrun and underrun protection.

Note that protection of the kernel text segment—and the text segments of kernel modules dynamically loaded into the kernel space—is not provided by default. On the other hand, the text segment of RTPs and shared libraries is always write-protected, whether or not VxWorks is configured with the **INCLUDE_PROTECT_TEXT** component. Similarly, the execution stack of an RTP task is not affected by the **INCLUDE_PROTECT_TASK_STACK** or **INCLUDE_TASK_STACK_NO_EXEC** components—it is always protected unless the task is spawned with the **taskSpawn( )** option **VX_NO_STACK_PROTECT**.

Stack Overrun and Underrun Detection

VxWorks can be configured so that guard zones are inserted at the beginning and end of task execution stacks. For more information, see *Task Stack Guard Zones*, p.96.

The operating system can also be configured to insert guard zones at both ends of the interrupt stack. For more information, see *Interrupt Stack Protection*, p.165.

Non-Executable Task Stack

VxWorks can be configured so that task stacks are non-executable. For more information, see *Configuring VxWorks for Additional Memory Protection*, p.272 and *Non-Executable Kernel Task Stacks*, p.97.

Text Segment Write Protection

All text segments are write-protected when VxWorks is configured with the **INCLUDE_PROTECT_TEXT** component. When VxWorks is loaded, all text segments are write-protected The text segments of any additional object modules loaded in the kernel space using **ld( )** are automatically marked as read-only. When object modules are loaded, memory that is to be write-protected is allocated in page-size increments. No additional steps are required to write-protect kernel application code.

Exception Vector Table Write Protection

When VxWorks is configured with the **INCLUDE_PROTECT_VEC_TABLE** component, the exception vector table is write-protected during system initialization.

The architecture-specific API provided to modify the vector table automatically write-enables the exception vector table for the duration of the call. For more information about these APIs, see the *VxWorks Architecture Supplement* for the architecture in question.

10.13 Memory Error Detection

Support for memory error detection is provided by the **memEdrLib** library and the Run-Time Error Checker (RTEC) facility.

The **memEdrLib** library performs error checks of operations in the user heap and memory partitions in an RTP. This library can be linked to an executable compiled with either the Wind River Compiler or the GNU compiler.

The Run-Time Error Checker (RTEC) facility checks for additional errors, such as buffer overruns and underruns, static and automatic variable reference checks.

NOTE: RTEC is only provided by the Wind River Diab Compiler, and it is the only feature of the compiler's Run-Time Analysis Tool Suite (RTA) that is supported for VxWorks. RTEC is therefore available only for processor architectures that are supported by the Diab compiler (and not the x86 and x86-64 processor architectures).

Errors detected by these facilities are reported by the error detection and reporting facility, which must, therefore be included in the VxWorks kernel configuration. See *13. Error Detection and Reporting*.

NOTE: The memory error detection facilities described in this section are not included in any user-mode shared library provided by Wind River. They can only be statically linked with RTP application code.

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 added to VxWorks to perform automatic, programmatic, and interactive error checks on **memLib** and **memPartLib** operations.

The instrumentation provided by **memEdrLib** 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, they help detect bounds-check violations, buffer over-runs and under-runs, pointer references to free memory blocks, pointer references to automatic variables outside the scope of the variable, and so on. Note that compiler-assisted instrumentation must be used in order to track buffer underruns and overruns. For information about compiler instrumentation, see *Compiler Instrumentation for 32-Bit VxWorks with RTEC*, p.282.

Errors detected by the automatic checks are logged by the error detection and reporting facility.

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. Detecting this type of errors depends on the compiler assisted instrumentation (see *Compiler Instrumentation for 32-Bit VxWorks with RTEC*, p.282).
- Allocation returns block address that is in the task's stack space. This would indicate corruption in the partition data structures.
- Allocation returns block address that is in the kernel's static data section. This would indicate corruption in the partition data structures.
- Freeing a pointer that is in the task's stack space.
- Freeing memory that was already freed and is still in the free queue.
- Freeing memory that is in the kernel's static data section. Detecting this type of errors depends on the compiler assisted instrumentation (see *Compiler Instrumentation for 32-Bit VxWorks with RTEC*, p.282).
- Freeing memory in a different partition than the one in which it was allocated.
- Freeing a partial memory block.
- Freeing a memory block with the guard zone corrupted, when the **MEDR_BLOCK_GUARD_ENABLE** environment variable is **TRUE**.
- Pattern in a memory block which is in the free queue has been corrupted, when the **MEDR_FILL_FREE_ENABLE** environment variable is **TRUE**.

Shell Commands

The show functions and commands described in Table 10-9 are available for use with the shell's C and command interpreters to display information.

Table 10-9 **Shell Commands**

C Interpreter	Command Interpreter	Description
edrShow( )	**edr show**	Displays error records.
memEdrPartShow( )	**mem part list**	Displays a summary of the instrumentation information for memory partitions in the kernel.
memEdrBlockShow( )	**mem block list**	Displays information about allocated blocks. Blocks can be selected using a combination of various querying criteria: partition ID, block address, allocating task ID, block type.

Table 10-9 **Shell Commands** (cont'd)

C Interpreter	Command Interpreter	Description
memEdrFreeQueueFlush( )	**mem queue flush**	Flushes the free queue. When this function is called, freeing of all blocks in the free queue is finalized so that all corresponding memory blocks are returned the free pool of the respective partition.
memEdrBlockMark( )	**mem block mark** and **mem block unmark**	Marks or unmarks selected blocks allocated at the time of the call. The selection criteria may include partition ID and/or allocating task ID. This function can be used to monitor memory leaks by displaying information of unmarked blocks with **memBlockShow( )** or **mem block list**.

Configuration for Kernel Memory Partition and Heap Instrumentation

To enable the basic level of memory partition and heap instrumentation in the kernel, the following components must be included into the kernel configuration:

- **INCLUDE_MEM_EDR**, includes the basic memory partition debugging functionality and instrumentation code.
- **INCLUDE_EDR_ERRLOG**, **INCLUDE_EDR_POLICIES** and **INCLUDE_EDR_SHOW** for error detection, reporting, and persistent memory. For more information see *13. Error Detection and Reporting*.

The following component may also be included:

- **INCLUDE_MEM_EDR_SHOW**, for enabling the show functions.

Configuration Parameters

The following parameters of the **INCLUDE_MEM_EDR** component can be modified:

MEDR_EXTENDED_ENABLE
Set to **TRUE** to enable logging trace information for each allocated block, but at the cost of increased memory used to store entries in the allocation database. The default setting is **FALSE**.

MEDR_FILL_FREE_ENABLE
Set to **TRUE** to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. The default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN
Maximum length of the free queue. When a memory block is freed, instead of immediately returning it to the partition's memory pool, it is kept in a queue.

This is useful for detecting references to a memory block after it has been freed. When the queue reaches the maximum length allowed, the blocks are returned to the respective memory pool in a FIFO order. Queuing is disabled when this parameter is 0. Default setting for this parameter is 64.

MEDR_BLOCK_GUARD_ENABLE
Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, underruns, and some heap memory corruption. The default setting is **FALSE**.

MEDR_POOL_SIZE
Set the size of the memory pool used to maintain the memory block database.The default setting in the kernel is 1MB. The database uses 32 bytes per memory block without extended information enabled, and 64 bytes per block with extended information enabled (call stack trace). This pool is allocated from the kernel heap.

Kernel Code Example

The following kernel application code is used to demonstrate various errors detected with the heap and partition memory instrumentation. Its use is illustrated in *Shell Session Example*, p.277.

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

void heapErrors (void)
    {
    char * pChar;

    pChar = malloc (24);
    free (pChar + 2);         /* free partial block */
    free (pChar);

    free (pChar);             /* double-free block */
    pChar = malloc (32);      /* leaked memory */
    }
```

Shell Session Example

The following shell session is executed with the C interpreter. The sample code listed above is compiled and linked in the VxWorks kernel. The kernel must include the **INCLUDE_MEM_EDR** and **INCLUDE_MEM_EDR_SHOW** components. In order to enable saving call stack information, the parameter **MEDR_EXTENDED_ENABLE** is set **TRUE**. Also, the kernel should be configured with the error detection and reporting facility, including the show functions, as described in *13.2 Configuration for Error Detection and Reporting*, p.354.

First mark all allocated blocks:

```
-> memEdrBlockMark
    value = 6390 = 0x18f6
```

Next, clear the error log. This step is optional, and is done only to start with a clean log:

```
-> edrClear
    value = 0 = 0x0
```

The kernel application may be started in a new task spawned with the **sp( )** utility, as follows:

```
-> taskId = sp (heapErrors)
    New symbol "taskId" added to kernel symbol table.
    Task spawned: id = 0x246d010, name = t1
    taskId = 0x2469ed0: value = 38195216 = 0x246d010
```

At this point the application finished execution. The following command lists the memory blocks allocated, but not freed by the application task. Note that the listing shows the call stack at the time of the allocation:

```
-> memEdrBlockShow 0, 0, taskId, 5, 1

      Addr    Type    Size    Part ID  Task ID  Task Name       Trace
    -------- ------ -------- -------- -------- ------------ ------------
     246d7a0  alloc       32   269888  246d010          -t1 heapErrors()
                                                            memPartAlloc()
                                                            0x001bdc88()
```

Errors detected while executing the application are logged in persistent memory region.

Display the log using **edrShow( )**. The first error corresponds to line 9 in the test code; the second error corresponds to line 12.

```
-> edrShow
    ERROR LOG
    =========
    Log Size:         524288 bytes (128 pages)
    Record Size:      4096 bytes
    Max Records:      123
    CPU Type:         0x5a
    Errors Missed:    0 (old) + 0 (recent)
    Error count:      2
    Boot count:       20
    Generation count: 94

    ==[1/2]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          20
    OS Version:          6.0.0
    Time:                THU JAN 01 00:00:31 1970 (ticks = 1880)
    Task:                "t1" (0x0246d010)

    freeing part of allocated memory block
        PARTITION: 0x269888
        PTR=0x246bea2
        BLOCK: allocated at 0x0246bea0, 24 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : heapErrors ()
    0x00111364 heapErrors   +0x24 : free ()
    0x001c26f8 memPartFree  +0xa4 : 0x001bdbb4 ()
    0x001bdc6c memEdrItemGet+0x588: 0x001bd71c ()

    ==[2/2]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          20
    OS Version:          6.0.0
    Time:                THU JAN 01 00:00:31 1970 (ticks = 1880)
    Task:                "t1" (0x0246d010)

    freeing memory in free list
        PARTITION: 0x269888
        PTR=0x246bea0
        BLOCK: free block at 0x0246bea0, 24 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : heapErrors ()
    0x00111374 heapErrors   +0x34 : free ()
    0x001c26f8 memPartFree  +0xa4 : 0x001bdbb4 ()
    0x001bdc6c memEdrItemGet+0x588: 0x001bd71c ()
    value = 0 = 0x0
```

RTP Heap Instrumentation

A VxWorks kernel configured for RTP 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 functions 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 Diab or GNU toolchain):

-Wl,-umemEdrEnable

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 *RTPs and Environment Variables*, p.19, *3.2 RTP Application Structure*, p.26, and the API reference for **rtpSpawn( )**.

MEDR_EXTENDED_ENABLE
Set to **TRUE** to enable logging trace information for each allocated block, but at the cost of increased memory used to store entries in the allocation database. The default setting is **FALSE**.

MEDR_FILL_FREE_ENABLE
Set to **TRUE** to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. The default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN
Maximum length of the free queue. When a memory block is freed, instead of immediately returning it to the partition's memory pool, it is kept in a queue. This is useful for detecting references to a memory block after it has been freed. When the queue reaches the maximum length allowed, the blocks are returned to the respective memory pool in a FIFO order. Queuing is disabled when this parameter is 0. Default setting for this parameter is 64.

MEDR_BLOCK_GUARD_ENABLE
Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, underruns, and some heap memory corruption. The default setting is **FALSE**.

MEDR_POOL_SIZE
Set the size of the memory pool used to maintain the memory block database. The 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). This pool is allocated from the kernel heap.

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 functions communicate with a dedicated task in the process with message queues. The default setting is **FALSE**.

RTP Application 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.280.

Note that if you start the RTP application programmatically (as opposed from the shell), you must set the required environment variables programmatically as well. For information in this regard, see *RTPs and Environment Variables*, p.19 and the API reference entries for **setenv( )** and **putenv( )**.

```
#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 */
    }
```

Add the component **INCLUDE_EDR_SYSDBG_FLAG** in the VxWorks Kernel configuration.

Add a call to **memOptionsSet** from **usrAppInit**, as in this example:

```
memOptionsSet (MEM_ALLOC_ERROR_LOG_FLAG |
                   MEM_ALLOC_ERROR_EDR_WARN_FLAG |
                   MEM_BLOCK_CHECK |
                   MEM_BLOCK_ERROR_LOG_FLAG |
                   MEM_BLOCK_ERROR_EDR_FATAL_FLAG);
```

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
```

Compiler Instrumentation for 32-Bit VxWorks with RTEC

Additional errors are detected if the application is compiled using the Run-Time Error Checker (RTEC) feature of the Wind River Diab Compiler (for 32-bit VxWorks). The following flag should be used:

-Xrtc=*option*

NOTE: RTEC is only provided by the Wind River Diab Compiler, and it is the only feature of the compiler's Run-Time Analysis Tool Suite (RTA) that is supported for VxWorks. RTEC is therefore available only for processor architectures that are supported by the Diab compiler (and not the x86 and x86-64 processor architectures).

CAUTION: The RTEC facility only detects errors in code that is compiled with the Wind River Diab compiler **-Xrtc** flag. If calls are made to code that is not compiled with **-Xrtc**, errors may not be detected.

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.

Supported -Xrtc Options

Table 10-10 lists the **-Xrtc** options that are supported. Using the **-Xrtc** flag without specifying any option implements all supported options. An individual option (or bitwise OR'd combinations of options) can be enabled using the following syntax:

-Xrtc=*option*

Table 10-10 **-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

Error and Warning Logging

The errors and warnings detected by the RTEC compile-in instrumentation are logged by the error detection and reporting facility (see *13. 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 Diab Compiler User's Guide*.

Shell Commands

The compiler provided instrumentation automatically logs errors detected in applications using the error detection and reporting facility. For a list of the shell commands available for error logs see *13.4 APIs for Displaying and Clearing Error Records*, p.357.

RTEC Use for Kernel Code

Configuring VxWorks for Kernel RTEC Support

Support for this feature in the kernel is enabled by adding the **INCLUDE_MEM_EDR_RTC** component, as well as the components described in section *Configuration for Kernel Memory Partition and Heap Instrumentation*, p.276.

Kernel Code Example

Kernel application code built with the RTEC instrumentation has compiler-generated constructors. To ensure that the constructors are called when a module is dynamically downloaded, the module must be processed similarly to a C++ application. For example, the following make rule can be used:

```
TGT_DIR=$(WIND_BASE)/target
```

```
%.out : %.c
    @ $(RM) $@
    $(CC) $(CFLAGS) -Xrtc=0xfb $(OPTION_OBJECT_ONLY) $<
    @ $(RM) ctdt_$(BUILD_EXT).c
    $(NM) $(basename $@).o | $(MUNCH) > ctdt_$(BUILD_EXT).c
    $(MAKE) CC_COMPILER=$(OPTION_DOLLAR_SYMBOLS) ctdt_$(BUILD_EXT).o
    $(LD_PARTIAL) $(LD_PARTIAL_LAST_FLAGS) $(OPTION_OBJECT_NAME)$@ $(basename
$@).o ctdt_$(BUILD_EXT).o

include $(TGT_DIR)/h/make/rules.library
```

The the following application code generates various errors that can be recorded and displayed. Its use is illustrated in *Shell Session Example for Kernel Code*, p.284.

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

void refErrors ()
    {
    char name[] = "very_long_name";
    char * pChar;
    int state[] = { 0, 1, 2, 3 };
    int ix = 0;

    pChar = malloc (13);
    memcpy (pChar, name, strlen (name)); /* bounds check violation */

    /* of allocated block */

    for (ix = 0; ix < 4; ix++)
        state[ix] = state [ix + 1];      /* bounds check violation */

    free (pChar);

    memcpy (pChar, "another_name", 12);  /* reference a free block */
    }
```

Shell Session Example for Kernel Code

The following shell session log is executed with the C interpreter. The kernel sample code listed above is compiled and linked in the VxWorks kernel. The kernel must include the **INCLUDE_MEM_EDR** and **INCLUDE_MEM_EDR_RTC** components. Also, the kernel should be configured with the error detection and reporting facility, including the show functions, as described in *13.2 Configuration for Error Detection and Reporting*, p.354.

First, clear the error log to start with a clean log:

```
-> edrClear
    value = 0 = 0x0
```

Start the kernel application in a new task spawned with the **sp( )** utility:

```
-> sp refErrors
    Task spawned: id = 0x246d7d0, name = t2
    value = 38197200 = 0x246d7d0
```

At this point the application finished execution. Errors detected while executing the application are logged in the persistent memory region. Display the log using **edrShow( )**. In the example below, the log display is interspersed with description of the errors.

```
-> edrShow
    ERROR LOG
    =========
    Log Size:         524288 bytes (128 pages)
    Record Size:      4096 bytes
    Max Records:      123
    CPU Type:         0x5a
    Errors Missed:    0 (old) + 0 (recent)
    Error count:      3
    Boot count:       21
    Generation count: 97
```

The first error corresponds to line 13 in the test code. A string of length 14 is copied into a allocated buffer of size 13:

```
==[1/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51738)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:13

    memory block bounds-check violation
        PTR=0x246be60  OFFSET=0  SIZE=14
        BLOCK: allocated at 0x0246be60, 13 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x00111470 refErrors    +0xe4 : __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

The second error refers to line 18: the local **state** array is referenced with index 4. Since the array has only four elements, the range of valid indexes is 0 to 3:

```
==[2/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51738)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:18

    memory block bounds-check violation
        PTR=0x278ba94  OFFSET=16  SIZE=4
        BLOCK: automatic at 0x0278ba94, 16 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x001114a0 refErrors    +0x114: __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

The last error is caused by the code on line 22. A memory block that has been freed is being modified:

```
==[3/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51739)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:22

    pointer to free memory block
        PTR=0x246be60  OFFSET=0  SIZE=12
        BLOCK: free block at 0x0246be60, 13 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x00111518 refErrors    +0x18c: __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

RTEC Use for RTP Application Code

Configuring VxWorks for RTP RTEC Support

Support for this feature for RTPs is enabled by configuring VxWorks with the basic error detection and reporting facilities. See *13.2 Configuration for Error Detection and Reporting*, p.354.

RTP Application Code Example

This RTP 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 for RTP Application Code*, p.286.

```
#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 for RTP Application Code

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
```

11
I/O System

11.1 About the VxWorks I/O System

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

This includes:

- 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.

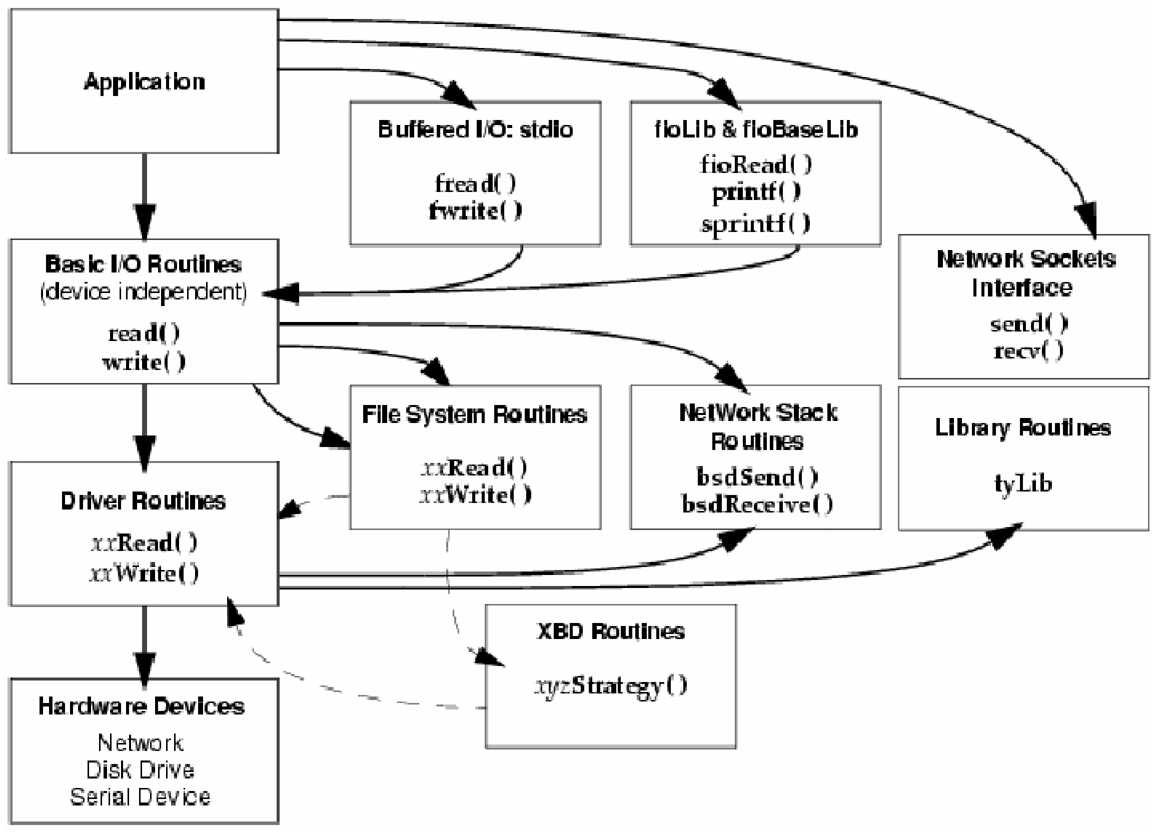
For information about VxWorks devices, see *12. Devices*.

Kernel I/O

Internally, the VxWorks I/O system has a unique design that makes it faster and more flexible than most other I/O systems. These are important attributes in a real-time system.

The diagram in Figure 11-1 illustrates the relationships between the different elements of the VxWorks kernel I/O system. All of these elements are discussed in this chapter, except for file system functions (which are dealt with in the *VxWorks 7 File Systems Programmer's Guide*), and the network elements (which are covered in the *Wind River Network Stack Programmer's Guide*).

Figure 11-1 **Kernel I/O System**

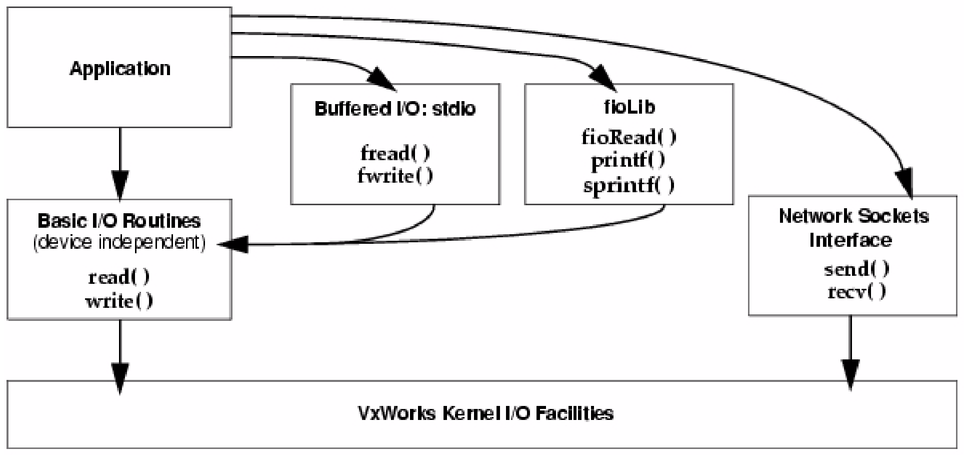


NOTE: The dotted lines in Figure 11-1 indicate that the XBD facility is required for some file systems, but not others. For example, HRFS and dosFs require XBD, while ROMFS has its own interface to drivers. See *12.9 Extended Block Device Facility: XBD*, p.329.

I/O System and RTP Applications

The diagram in Figure 11-2 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 11-2 **I/O System for RTPs (Processes)**



Differences Between VxWorks and Host System I/O

Most commonplace uses of I/O in VxWorks are completely source-compatible with I/O in UNIX and Windows. There are, however, differences in the following areas:

- File Descriptors
- I/O Control
- Device Configuration
- Device Driver Functions

In VxWorks, file descriptors are unique to the kernel and to each process—as in UNIX and Windows. The kernel and each process has its own universe of file descriptors, distinct from each other. When the process is created, its universe of file descriptors is initially populated by duplicating the file descriptors of its creator. (This applies only when the creator is a process. If the creator is a kernel task, only the three standard I/O descriptors 0, 1 and 2 are duplicated.) Thereafter, all open, close, or *dup* activities affect only that process' universe of descriptors. In kernel and in each process, file descriptors are global to that entity, meaning that they are accessible by any task running in it. In the kernel, however, standard input, standard output, and standard error (0, 1, and 2) can be made task specific. For more information see *File Descriptors*, p.296 and *Standard I/O Redirection in the Kernel*, p.297.

The specific parameters passed to **ioctl( )** functions may differ between UNIX and VxWorks.

In the VxWorks kernel, device drivers can be installed and removed dynamically. But only in the kernel space.

In UNIX, device drivers execute in system mode and cannot be preempted. In VxWorks, driver functions can be preempted because they execute within the context of the task that invoked them.

11.2 VxWorks I/O Facility Configuration

Table 11-1 **Primary I/O Components**

Component	Description
INCLUDE_IO_BASIC	Basic I/O functionality.
INCLUDE_IO_FILE_SYSTEM	File system support.
INCLUDE_POSIX_DIRLIB	POSIX directory utilities.
INCLUDE_IO_REMOVABLE	support for removable file systems.
INCLUDE_IO_POSIX	POSIX I/O support.
INCLUDE_IO_RTP	I/O support for RTPs.
INCLUDE_IO_MISC	Miscellaneous IO functions that are no longer referenced but are provided for backwards compatibility.
INCLUDE_POSIX_AIO	Asynchronous I/O support.
INCLUDE_POSIX_AIO_SHOW	Asynchronous I/O show function.
INCLUDE_POSIX_UMASK	**umask( )**.

Additional Components

The **INCLUDE_IO_SYSTEM** component is provided for backward compatibility. It includes all the components listed above.

Components that provide support for additional features are described throughout this chapter.

11.3 I/O Devices, Named Files, and File Systems

VxWorks applications access I/O devices by opening named files.

A named file can refer to either of the following:

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

I/O can be done to or from either type of named file in the same way. For VxWorks, they are all called *files*, even though they refer to very different physical objects.

Device Naming

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.

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. For more information, see *12. Devices*.

File Names and Device Names

A filename is specified as a character string. An unstructured device is specified with the device name. For structured devices with file systems, the device name is followed by a filename. For example:

/pipe/mypipe
Refers to a named pipe—by convention, pipe names begin with **/pipe**.

/tyCo/0
Refers to a serial channel.

/usr/myfile
Refers to a file called **myfile**, on the disk device **/usr**.

/ata0:0/file1
Refers to the file **file1** on the **/ata0:0** disk device.

Default I/O Device for Files

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

For example, if the file name is **mars:/usr/xeno/foo/myfile**, the I/O function is directed to the remote machine named **mars**. (For information about the use remote system prefixes, see *11.4 Remote File System Access From VxWorks*, p.294.) If a matching device name cannot be found, then the I/O function is directed at a *default device*.

You can set the default device to any device in the system or no device at all—in which case failure to match a device name returns an error. To obtain the current default path, use **getcwd( )**. To set the default path, use **chdir( )**.

File I/O

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. For more information, see *11.5 Basic I/O*, p.295 and *11.6 Standard I/O*, p.308.

11.4 Remote File System Access From VxWorks

Accessing remote file systems from VxWorks—either programmatically or from the shell—requires special consideration, depending on the connection protocol.

NFS File System Access from VxWorks

By convention, NFS-based network devices are *mounted* with names that begin with a slash. To access directories and files from VxWorks, use the path from the mount point. For example, if the mount point for **/usr/xeno/foo** on VxWorks is **/foo**, then **/usr/xeno/foo/myFile** is accessed from VxWorks as:

```
/foo/myFile
```

For example:

```
fd=open("/foo/myFile", O_CREAT | O_RDWR, 0);
```

For information about NFS, see the *VxWorks 7 Programmer's Guide: Network File System*.

Non-NFS Network File System Access from VxWorks WIth FTP or RSH

To access non-NFS file systems over a network with FTP or RSH, the name of the remote machine must be specified by prefixing the file system path with the remote machine name followed by a colon.

The exception to this rule is if the remote machine is a Linux host system, the host and colon prefix is not required (but may still be used). By default VxWorks uses the host system as the default device if it cannot find a matching device name, which works for Linux hosts. It does not work, however, with a Windows machine because VxWorks interprets Windows drive letters as device names—and they must therefore be disambiguated with the host name and colon prefix.

For example, the file **/usr/xeno/foo/myfile** on the Linux machine **mars** and the file **C:\bar\myotherfile** on Windows machine **jupiter** would be identified as follows:

```
    mars:/usr/xeno/foo/myfile

    jupiter:C:\bar\myotherfile
```

And the paths used accordingly:

```
    fd=open("mars:/usr/xeno/foo/myfile", O_CREAT | O_RDWR, 0);

    fd=open("jupiter:C:\bar\myotherfile", O_CREAT | O_RDWR, 0);
```

If **mars** is the host machine (for Linux), however, the following usage—without the machine name and colon—would also work:

```
    fd=open("/usr/xeno/foo/myfile", O_CREAT | O_RDWR, 0);
```

Note that for the files of a remote host to be accessible with RSH or FTP, permissions and user identification must be established on both the remote and local systems.

Remote System Name Definition

For the host system, the name is defined with the **host name** boot parameter (for which the default is simply **host**). For other remote machines, a network device must first be created by calling the kernel function **netDevCreate( )**.

Host File System Access from the VxWorks Simulator

The host file system is accessed from the VxWorks simulator by way of the pass-through file system (passFs).

For a Linux host, the path must be prefixed with name of this host on which the simulator is running, followed by a colon. For example:

```
mars:/usr/xeno/foo/myfile
```

For a Windows hosts, the path must be prefixed with literal string **host** followed by a colon. For example:

```
host:/C/bar/myotherfile
```

A colon may be used after the Windows drive letter, but this use is deprecated and is retained only for backward-compatibility reasons.

NOTE: Do not confuse the **host:** prefix used with the VxWorks simulator on a Windows host with the default host device name. The default host name is specified with the **host name** boot loader parameter, and it appears as **host:** in the list generated by the VxWorks shell **devs** command.

NOTE: The VxWorks simulator's pass-through file system requires forward-slash path delimiters, even on a Windows host.

For more information about the VxWorks simulator, see the *Wind River VxWorks Simulator User's Guide*.

11.5 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.

Table 11-2 **Basic I/O Functions**

Function	Description
creat( )	Creates a file.
remove( )	Deletes a file.
open( )	Opens a file (optionally, creates a file if it does not already exist.)
close( )	Closes a file.
read( )	Reads a previously created or opened file.
write( )	Writes to a previously created or opened file.
ioctl( )	Performs special control functions on files.

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 In Kernel

The number of file descriptors available in the kernel is defined with the **NUM_FILES** configuration macro. This specifies the size of the file descriptor table, which controls how many file descriptors can be simultaneously in use. The default number is 50, but it can be changed to suit the needs of the system.

To avoid running out of file descriptors, and encountering errors on file creation, applications should close any descriptors that are no longer in use.

The size of the file descriptor table for the kernel can also be changed at programmatically. The **rtpIoTableSizeGet( )** function reads the size of the file descriptor table and the **rtpIoTableSizeSet( )** function changes it. Note that these functions can be used with both the kernel and processes (the I/O system treats the kernel as a special kind of process).

The calling entity—whether kernel or process—can be specified with an **rtpIoTableSizeSet( )** call by setting the first parameter to zero. The new size of the file descriptor table is set with the second parameter. Note that you can only increase the size.

File Descriptor Table in RTPs

In user space (RTPs) 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( )** function reads the current size of the table, and the **rtpIoTableSizeSet( )** function changes it.

By default, file descriptors are reclaimed only when the file is closed for the last time. However, the **dup( )** and **dup2( )** functions can be used to duplicate a file descriptor. For more information, see *Standard I/O Redirection in the Kernel*, p.297.

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**).

In the kernel, all tasks read their standard input—like **getchar( )**—from file descriptor 0. Similarly file descriptor 1 is used for standard output—like **printf( )**. And file descriptor 2 is used for outputting error messages. Using these descriptors, you can manipulate the input and output for many tasks at once by redirecting the files associated with the descriptors.

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

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

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.

Standard I/O Redirection in the Kernel

When VxWorks is initialized, the global standard I/O file descriptors, **stdin** (0), **stdout** (1) and **stderr** (2), are set in the kernel to the system console device file descriptor by default, which is usually the serial tty device.

Each kernel task uses these global standard I/O file descriptors by default. Thus, any standard I/O operations like calls to **printf( )** and **getchar( )** use the global standard I/O file descriptors.

Standard I/O can be redirected, however, either at the individual task level, or globally for the kernel.

The standard I/O of a specific task can be changed with the **ioTaskStdSet( )** function. The parameters of this function are the task ID of the task for which the change is to be made (0 is used for the calling task itself), the standard file descriptor to be redirected, and the file descriptor to direct it to. For example, a task can make the following call to write standard output to the **fileFd** file descriptor:

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

The third argument (**fileFd** in this case) can be any valid open file descriptor. If it is a file system file, all the task's subsequent standard output, such as that from **printf( )**, is written to it.

To reset the task standard I/O back to global standard I/O, the third argument can be 0, 1, or 2.

The global standard I/O file descriptors can also be changed from the default setting, which affects all kernel tasks except that have had their task-specific standard I/O file descriptors changed from the global ones.

Global standard I/O file descriptors are changed by calling **ioGlobalStdSet( )**. The parameters to this function are the standard I/O file descriptor to be redirected, and the file descriptor to direct it to. For example:

```
ioGlobalStdSet (1, newFd);
```

This call sets the global standard output to **newFd**, which can be any valid open file descriptor. All tasks that do not have their individual task standard output redirected are affected by this redirection, and all their subsequent standard I/O output goes to **newFd**.

The current settings of the global and any task's task standard I/O can be determined by calling **ioGlobalStdGet( )** and **ioTaskStdGet( )**. For more information, see the VxWorks API references for these functions.

Issues with Standard I/O Redirection in the Kernel

Be careful with file descriptors used for task standard I/O redirection to ensure that data corruption does not occur. Before any task’s standard I/O file descriptors are closed, they should be replaced with new file descriptors with a call to **ioTaskStdSet( )**.

If a task’s standard I/O is set with **ioTaskStdSet( )**, the file descriptor number is stored in that task’s memory. In some cases, this file descriptor may be closed, released by some other task or the one that opened it. Once it is released, it may be reused and opened to track a different file. Should the task holding it as a task standard I/O descriptor continue to use it for I/O, data corruption is unavoidable.

As an example, consider a task spawned from a telnet or rlogin session. The task inherits the network session task's standard I/O file descriptors. If the session exits, the standard I/O file descriptors of the network session task are closed. However, the spawned task still holds those file descriptors as its task standard I/O continued with input and output to them. If the closed file descriptors are recycled and re-used by other **open( )** call, however, data corruption results, perhaps with serious consequences for the system. To prevent this from happening, all spawned tasks must have their standard I/O file descriptors redirected before the network session is terminated.

The following example illustrates this scenario, with redirection of a spawned task’s standard I/O to the global standard I/O from the shell before logout. The **taskspawn( )** call is abbreviated to simplify presentation.

```
-> taskSpawn "someTask",......
Task spawned: id = 0x52a010, name = t4
value = 5414928 = 0x52a010
-> ioTaskStdSet 0x52a010,0,0
```

```
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,1,1
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,2,2
value = 0 = 0x0
-> logout
```

The next example illustrates task standard I/O redirection to other file descriptors.

```
-> taskSpawn "someTask",......
Task spawned: id = 0x52a010, name = t4
value = 5414928 = 0x52a010
-> ioTaskStdSet 0x52a010,0,someOtherFdx
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,1,someOtherFdy
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,2,someOtherFdz
value = 0 = 0x0
-> logout
```

Standard I/O Redirection in RTPs

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.

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( )** functions 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 function is used to save the original file descriptors so they can be restored later. The second function 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 functions are used.

First use the **dup( )** function 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( )**.

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( )** function—or **creat( )**, as discussed in the next section. The arguments to **open( )** are the filename, the type of access, and the mode (file permissions):

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

For **open( )** calls made in the kernel, the mode parameter can be set to zero if file permissions do not need to be specified.

For **open( )** calls made in processes, the mode parameter is optional.

For both user-level and kernel calls, the file-access options that can be used with the *flags* parameter to **open( )** are listed in Table 11-3.

Table 11-3 **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.
O_RSYNC	Read on the file descriptor complete at the same sync level as **O_DSYNC** and **O_SYNC** flags.

Table 11-3 **File Access Options** (cont'd)

Flag	Description
O_APPEND	Set the file offset to the end of the file prior to each write, which guarantees that writes are made at the end of the file. It has no effect on devices other than the regular file system.
O_NONBLOCK	Non-blocking I/O.
O_NOCTTY	If the named file is a terminal device, don't make it the controlling terminal for the process.
O_TRUNC	Open with truncation. If the file exists and is a regular file, and the file is successfully opened, its length is truncated to 0. It has no effect on devices other than the regular file system.

WARNING: While the third parameter to **open( )**—*mode*, for file permissions—is usually optional for other operating systems, it is required for the VxWorks implementation of **open( )** in the kernel (but not in processes). When the mode parameter is not appropriate for a given call, it should be set to zero. Note that this can be an issue when porting software from UNIX to VxWorks.

Note the following special cases with regard to use of the file access and mode (file permissions) parameters to **open( )**:

- In general, you can open only preexisting devices and files with **open( )**. However, with NFS network, dosFs, and HRFS devices, you can also create files with **open( )** by OR'ing **O_CREAT** with one of the other access flags.
- HRFS directories can be opened with the **open( )** function, 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( )** function, 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 RTPs (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 (*File Descriptors*, p.296), 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.

Since the kernel only terminates when the system shuts down, there is no situation analogous to file descriptors being closed automatically when a process terminates. File descriptors in the kernel can only be closed by direct command.

Create and Remove

File-oriented devices must be able to create and remove files as well as open existing files.

The **creat( )** function 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( )** function returns a file descriptor identifying the new file.

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

Note that with the HRFS file system the **creat( )** function 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( )** function is not POSIX-compliant and the second parameter is used for open mode flags.

The **remove( )** function 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( )** function performs the same function as **open( )**. The **remove( )** function, however has no effect.

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( )** function 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( )** function 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( )** function returns the number of bytes written; if the number returned is not equal to the number requested, an error has occurred.

The user-level (RTP) **read( )** and **write( )**functions are POSIX-compliant.

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( )** function 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( )** function 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.

I/O Control

The **ioctl( )** function 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( )** function are the file descriptor, a code that identifies the control function requested, and an optional function-dependent argument:

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

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

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

The discussion of each devices in *12. Devices* summarizes the **ioctl( )** functions available for that 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 user-level (RTP) **ioctl( )** function is POSIX-compliant. In addition, the *arg* parameter is optional. Both of the following are legitimate calls:

```
    fd = ioctl (fd, func, arg);
    fd = ioctl (fd, func);
```

VxWorks also provides the user-mode **posix_devctl( )** function for special devices. For more information, see the *VxWorks Application API Reference* entry.

Pending on Multiple File Descriptors with select( )

The VxWorks **select( )** function provides a UNIX- and Windows-compatible method for pending on multiple file descriptors (allowing tasks to wait for multiple devices to become active).

The kernel **selectLib** library includes **select( )** and related functions, and is provided with the **INCLUDE_SELECT** component.

For the user-level (RTP) **select( )** function, 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( )** function returns when one or more file descriptors are ready or a timeout has occurred. Using the **select( )** function, 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 11-4.

Table 11-4 **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( )**.

For information on writing a device driver that supports **select( )**, see *Implementing select( )*, p.346.

Example 11-1 **Using select( ) in the Kernel**

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

#include <vxWorks.h>
#include <selectLib.h>
#include <fcntl.h>

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

/************************************************************************
* selServer - reads data as it becomes available from two different pipes
*
* Opens two pipe fds, reading from whichever becomes available. The
* server code assumes the pipes have been created from either another
* task or the shell. To test this code from the shell do the following:
*  -> ld < selServer.o
*  -> pipeDevCreate ("/pipe/highPriority", 5, 1024)
*  -> pipeDevCreate ("/pipe/normalPriority", 5, 1024)
*  -> fdHi = open   ("/pipe/highPriority", 1, 0)
*  -> fdNorm = open ("/pipe/normalPriority", 1, 0)
*  -> iosFdShow
*  -> sp selServer
*  -> i

* At this point you should see selServer's state as pended. You can now
* write to either pipe to make the selServer display your message.
*  -> write fdNorm, "Howdy", 6
*  -> write fdHi, "Urgent", 7
*/

STATUS selServer (void)
    {
    struct fd_set readFds;      /* bit mask of fds to read from */
    int      fds[MAX_FDS];      /* array of fds on which to pend */
    int      width;             /* number of fds on which to pend */
    int      i;                 /* index for fd array */
    char     buffer[MAX_DATA];  /* buffer for data that is read */
/* open file descriptors */

    if ((fds[0] = open (PIPEHI, O_RDONLY, 0)) == ERROR)
        {
        close (fds[0]);
        return (ERROR);
        }
    if ((fds[1] = open (PIPENORM, O_RDONLY, 0)) == ERROR)
        {
        close (fds[0]);
        close (fds[1]);
        return (ERROR);
        }

/* loop forever reading data and servicing clients */

    FOREVER
        {
        /* clear bits in read bit mask */
        FD_ZERO (&readFds);


/* initialize bit mask */

        FD_SET (fds[0], &readFds);
        FD_SET (fds[1], &readFds);
        width = (fds[0] > fds[1]) ? fds[0] : fds[1];
        width++;

/* pend, waiting for one or more fds to become ready */

    if (select (width, &readFds, NULL, NULL, NULL) == ERROR)
        {
        close (fds[0]);
        close (fds[1]);
        return (ERROR);
        }

/* step through array and read from fds that are ready */

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

Example 11-2 **Using select( ) in RTP Applications**

```
/* 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);
}
```

POSIX File System Functions

VxWorks provides POSIX I/O and file system functions for various file manipulations, for both kernel and user (RTP) applications. These functions are described in Table 11-5.

Table 11-5 **POSIX File System Functions**

Function	Description
access( )	Determine accessibility of a file.
chmod( )	Change the permission mode of a file.
fchmod( )	Change the permission mode of a file.
fcntl( )	Perform control functions over open files.
fdatasync( )	Synchronize the data of a file.
fpathconf( )	Determine the current value of a configurable limit.
fsync( )	Synchronize a file.
link( )	Link a file.
pathconf( )	Determine the current value of a configurable limit.
rename( )	Change the name of a file.
umask( )	Set and get the file mode creation mask.
unlink( )	Unlink a file. User-level only.

For more information, see the API references for **fsPxLib**, **ioLib**, and **umaskLib**.

11.6 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 functions are provided by the Dinkum C libraries.

Configuring VxWorks With Standard I/O

For the VxWorks kernel, the majority of the traditional standard I/O functions are provided with one VxWorks component, and small subsets of functions with other components. This modular approach allows for reducing the footprint of systems that only require the most commonly used standard I/O functions—such as **printf( )**, **sprintf( )**, and **sscanf( )**—by only including the component (or components) that provide these functions.

The following components provide the standard I/O functions:

INCLUDE_ANSI_STDIO
Provides the **ansiStdio** library, which includes most ANSI standard I/O functions except **printf( )**, **sprintf( )**, and **sscanf( )**.

INCLUDE_ANSI_STDIO_EXTRA
Provides VxWorks extensions to ANSI standard I/O in **ansiStdio**.

INCLUDE_FORMATTED_OUT_BASIC
Provides the **fioBaseLib** library (unbuffered), which includes **printf( )**, **sprintf( )**, and a few related functions.

INCLUDE_FORMATTED_IO
Provides the **fioLib** library (unbuffered), which includes **sscanf( )** and a few related functions.

The functions in **fioBaseLib** and **fioLib** do not use the buffered I/O facilities provided by the standard I/O library **ansiStdio**. They can therefore be used even if **ansiStdio** has not been included in the configuration of VxWorks.

About the Kernel printf( ), sprintf( ), and scanf( ) Functions

The VxWorks kernel implementations of **printf( )**, **sprintf( )**, and **sscanf( )** comply with ANSI standards. Note, however, the following:

- The **printf( )** function is not buffered. The function is implemented in this way so that **fioBaseLib** library does not require the overhead of buffering facilities. The performance advantages of buffering are, in any case, largely irrelevant with regard to **printf( )**.
- Applications that require **printf( )**-style output that is buffered can call **fprintf( )** explicitly to *stdout*.
- While **sscanf( )** is provided by the **INCLUDE_FORMATTED_IO** component, **scanf( )** is provided by the **INCLUDE_ANSI_STDIO** component. In order to use **scanf( )**, you must therefore include the larger **ansiStdio** library provided by the **INCLUDE_ANSI_STDIO** component.

About Standard I/O and Buffering

The use of the buffered I/O functions provided with standard I/O can provide a performance advantage over the non-buffered basic I/O functions when an application performs many small read or write operations. (For information about non-buffered I/O, see *11.5 Basic I/O*, p.295).

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 function 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 functions and macros.

When a file is created or opened with a standard I/O function, it is commonly referred to as associating a *stream* with the file. To open a file, use **fopen( )**, as follows:

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

The **fopen( )** call returns a pointer to a **FILE** object (declared as **FILE** *), which is referred to as a *file pointer*. The **FILE** object contains information required by the standard I/O library to manage the stream, including a file descriptor that is actually used for I/O, information about the buffer used for the stream, and so on. (By contrast, the low-level I/O functions simply identify a file with a file descriptor, which is a small integer.)

NOTE: The user-mode (RTP) implementation of **fopen( )** has a limit of 20 file descriptors.

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 functions 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 functions 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.

NOTE: The VxWorks kernel implementation of **printf( )** is ANSI standard, but not buffered. This implementation allows for smaller footprint configurations of VxWorks that still provide key standard I/O functions (For more information, see *Configuring VxWorks With Standard I/O*, p.309). The performance advantages of buffering are, in any case, largely irrelevant with regard to **printf( )**.

About Standard Input, Standard Output, and Standard Error

As discussed in *11.5 Basic I/O*, p.295, 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.

11.7 Other Formatted I/O

VxWorks provides additional kernel formatted I/O facilities.

Output in Serial I/O Polled Mode: kprintf( )

The **kprintf( )** kernel function performs formatted output in a manner similar to **printf( )**, except that the output characters are sent to the target's serial port in polled mode. The function is designed to be used primarily as a debug facility. It can be used during the kernel boot sequence (before interrupts are enabled or the I/O system is initialized) and within ISRs, when it is otherwise impossible to print messages to the target console using **printf( )**.

For more information, see the **kprintf( )** API reference entry. The function is provided by the **INCLUDE_DEBUG_KPRINTF** component.

The **kputs( )** function provides similar functionality with un-formatted string output. For more information, see the API reference entry. The function is provided by the **INCLUDE_DEBUG_KPUTS** component.

The optional component **INCLUDE_DEBUG_KWRITE_USER** can be used to implement output to storage media; see *Writing to User-Defined Storage Media With kprintf( ) and kputs( )*, p.311.

Writing to User-Defined Storage Media With kprintf( ) and kputs( )

The kernel **kprintf( )** and **kputs( )** functions can be made to write to user-defined storage media (such as flash). The following function, for example, is used to implement writing to user-reserved memory:

```
#include <vxWorks.h>
#include <string.h>
#include <sysLib.h>
STATUS usrRtn
(
char * buffer,
size_t len
)
{
static off_t    offset;
char *        pDstAddr;
size_t         reservedSize;

userReservedGet (&pDstAddr, &reservedSize);
pDstAddr += offset;

if (reservedSize >= len)
    {
```

```
    bcopy (pBuffer, pDstAddr, len);
    offset += len;
    return OK;
    }
else
    return ERROR;}
```

To make use of this function, perform the following steps:

1. Add the code to a C file in any directory.
2. Create a VxWorks image project (VIP) for the appropriate BSP.
3. Add the C file to the project.
4. Add the **INCLUDE_DEBUG_KPRINTF**, **INCLUDE_DEBUG_KWRITE_USER**, and **INCLUDE_USER_RESERVED_MEMORY** components to the project.
5. Set the **DEBUG_KWRITE_USR_RTN** and **USER_RESERVED_MEM** configuration parameters, as appropriate.
6. Build the project and boot VxWorks.

When calls are made to **kprintf( )** and **kputs( )**, output is written to user-reserved memory. Messages are written one after the other, from the start of user reserved memory. Messages can be read by dumping memory from the start of user reserved memory.

Additional Formatted I/O Functions

The user-level **fioLib** library provides additional formatted output and scanning functions (non-ANSI).

The kernel **fioLib** and **fioBaseLib** libraries provide additional formatted but unbuffered output and scanning functions (non-ANSI).

In both cases, for example, the function **printErr( )** is analogous to **printf( )** but outputs formatted strings to the standard error file descriptor (2). The function **fdprintf( )** outputs formatted strings to a specified file descriptor.

Message Logging

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

The **logLib** facilities are provided by the **INCLUDE_LOGLIB** component.

11.8 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 currently included in any RTP shared library provided by Wind River. 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.

The POSIX AIO Functions

The VxWorks library **aioPxLib** provides POSIX AIO functions. To access a file asynchronously, open it with the **open( )** function, like any other file. Thereafter, use the file descriptor returned by **open( )** in calls to the AIO functions. The POSIX AIO functions (and two associated non-POSIX functions) are listed in Table 11-6.

The default VxWorks initialization code in the kernel calls **aioPxLibInit( )** automatically when the POSIX AIO component is included in VxWorks with **INCLUDE_POSIX_AIO**.

The **aioPxLibInit( )** function takes one parameter, the maximum number of **lio_listio( )** calls that can be outstanding at one time. By default this parameter is **MAX_LIO_CALLS**. When the parameter is 0 (the default), the value is taken from **AIO_CLUST_MAX** (defined in **aioPxLibP.h**).

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

Table 11-6 **Asynchronous Input/Output Functions**

Function	Description
aioPxLibInit( )	Initializes the AIO library (non-POSIX; kernel). Kernel only.
aioShow( )	Displays the outstanding AIO requests (non-POSIX). Kernel only.
aio_read( )	Initiates an asynchronous read operation.

Table 11-6 **Asynchronous Input/Output Functions** (cont'd)

Function	Description
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.

The kernel function **aioSysInit( )** takes three parameters: the number of AIO system tasks to spawn, and the priority and stack size for these system tasks. The number of AIO system tasks spawned equals the number of AIO requests that can be handled in parallel. The default initialization call uses three constants: **MAX_AIO_SYS_TASKS**, **AIO_TASK_PRIORITY**, and **AIO_TASK_STACK_SIZE**.

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

Table 11-7 lists the names of the constants, and shows the constants used within initialization functions when the parameters are left at their default values of 0, and where these constants are defined.

Table 11-7 **AIO Initialization Functions and Related Constants**

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

AIO Control Block

Each of the AIO calls takes an AIO control block (**aiocb**) as an argument. The calling function 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 function 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.

Using AIO

The functions **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 functions 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 function 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 functions that you can use to get information about the success or failure of the I/O operation: **aio_error( )** and **aio_return( )**. You can use **aio_error( )** to get the status of an AIO operation (success, failure, or in progress), and **aio_return( )** to obtain the return values from the individual I/O operations. Until an AIO operation completes, its error status is **EINPROGRESS**. To cancel an AIO operation, call **aio_cancel( )**. To force all I/O operations to the synchronized I/O completion state, use **aio_fsync( )**.

AIO with Periodic Checks for Completion

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

Example 11-3 **Asynchronous I/O**

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

/* includes */

#include <vxWorks.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <aio.h>

/* defines */

#define BUFFER_SIZE 200

struct aiocb    aiocb_read; /* read aiocb */
struct aiocb    aiocb_write; /* write aiocb */

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

STATUS aioExample (const char *exFile)
    {
    int            fd;
    char           buffer [BUFFER_SIZE]; /* buffer for read aiocb */
    static char *  test_string = "testing 1 2 3";
    int            error;

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


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

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

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

    /* initiate the read */
    if (aio_read (&aiocb_read) == -1)
    printf ("aioExample: aio_read failed\n");
    /* verify that it is in progress */
    if (aio_error (&aiocb_read) == EINPROGRESS)
    printf ("aioExample: read is still in progress\n");

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

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

    printf ("aioExample: error = %d\n", error);

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

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

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

Alternatives for Testing AIO Completion

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

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

The following example of kernel code is similar to the preceding **aioExample( )**, except that it uses signals for notification that the write operation has finished. If you test this from the shell, spawn the function to run at a lower priority than the AIO system tasks to assure that the test function does not block completion of the AIO request.

Example 11-4 **Asynchronous I/O with Signals**

```
#include <vxWorks.h>
#include <stdio.h>
#include <aio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

/* defines */

#define BUFFER_SIZE   200
#define LIST_SIZE    1
#define WRITE_EXAMPLE_SIG_NO  25 /* signal number */
#define READ_EXAMPLE_SIG_NO  26 /* signal number */

/* forward declarations */

void writeSigHandler (int sig, struct siginfo * info, void * pContext);
void readSigHandler (int sig, struct siginfo * info, void * pContext);
```

```
static struct aiocb         aiocb_read;   /* read aiocb  */
static struct aiocb         aiocb_write;  /* write aiocb */
static struct sigaction     write_action; /* signal info */
static struct sigaction     read_action;  /* signal info */
static char                 buffer [BUFFER_SIZE]; /* aiocb read buffer */

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

STATUS aioExampleSig (const char *exFile)
    {
    int            fd;
    static char *  test_string = "testing 1 2 3";

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

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

    /* set up signal handler for WRITE_EXAMPLE_SIG_NO */

    write_action.sa_sigaction = writeSigHandler;
    write_action.sa_flags = SA_SIGINFO;
    sigemptyset (&write_action.sa_mask);
    sigaction (WRITE_EXAMPLE_SIG_NO, &write_action, NULL);

    /* set up signal handler for READ_EXAMPLE_SIG_NO */

    read_action.sa_sigaction = readSigHandler;
    read_action.sa_flags = SA_SIGINFO;
    sigemptyset (&read_action.sa_mask);
    sigaction (READ_EXAMPLE_SIG_NO, &read_action, NULL);

    /* initialize read and write aiocbs */

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

    /* set up signal info */

    aiocb_read.aio_sigevent.sigev_signo = READ_EXAMPLE_SIG_NO;
    aiocb_read.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aiocb_read.aio_sigevent.sigev_value.sival_ptr =
    (void *) &aiocb_read;

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

    /* set up signal info */

    aiocb_write.aio_sigevent.sigev_signo = WRITE_EXAMPLE_SIG_NO;
    aiocb_write.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aiocb_write.aio_sigevent.sigev_value.sival_ptr =
    (void *) &aiocb_write;
    /* initiate the read */

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

    /* verify that it is in progress */

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

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

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

    close (fd);
    return (OK);
    }

void writeSigHandler
    (
    int                 sig,
    struct siginfo *    info,
    void *              pContext
    )
    {
    /* print out what was written */
    printf ("writeSigHandler: Got signal for aio write\n");

    /* write is complete so let's do cleanup for it here */
    if (aio_return (info->si_value.sival_ptr) == -1)
    {
    printf ("writeSigHandler: aio_return for aiocb_write failed\n");
    }
    }

void readSigHandler
    (
    int                 sig,
    struct siginfo *    info,
    void *              pContext
    )
    {
    /* print out what was read */
    printf ("readSigHandler: Got signal for aio read\n");

    /* write is complete so let's do cleanup for it here */
    if (aio_return (info->si_value.sival_ptr) == -1)
    {
    printf ("readSigHandler: aio_return for aiocb_read failed\n");
    }
    else
    {
    printf ("aioExample: message = %s\n", buffer);
    }
    }
```

12
Devices

12.1 About VxWorks Devices

The VxWorks I/O system allows different device drivers to handle the seven basic I/O functions. VxWorks provides serial devices (pseudo and pseudo-terminal), pipes, a pseudo I/O device for accessing memory directly, null devices, and block devices. In the kernel it provides support for PCMCIA and PCI. In addition, the extended block device (XBD) layer, provides an interface between block device drivers and file systems.

VxWorks provides NFS and non-NFS network devices and sockets, which are described briefly in this chapter, and in more detail in the *VxWorks 7 File Systems Programmer's Guide: Network File System* and the *Wind River Network Stack Programmer's Guide*.

For information about the I/O system itself, see *11. I/O System*.

12.2 VxWorks Devices

Table 12-1 **Devices Provided with VxWorks**

Device	Driver Description
tty	Terminal device
pty	Pseudo-terminal device
pipe	Pipe device
mem	RAM device
nfs	NFS client device
net	Network device for remote file access
null	Null device
ram	RAM device for creating a RAM disk
romfs	ROMFS device
–	Other hardware-specific device

12.3 Device Names

Note that the default device name length is 10 characters. If additional characters are used, they are truncated.

The VSB option **MAX_DEVNAME** allows you to set the maximum length of names for devices from 5 to 50 characters (the default setting is 10). If a device name exceeds the length defined by **MAX_DEVNAME**, the extra characters are truncated. The **NULL** character is not used to designate the end of the device name string, so you should use string-safe functions—such as **strncpy( )**, **strncat( )**, and **strncmp( )**—with device names.

CAUTION: Devices should not be given the same name, or they will overwrite each other in core I/O.

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.

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 the discussion of the virtual root file system (VRFS) in the *VxWorks 7 File Systems Programmer Guide*.

NOTE: Only VxBus-compatible drivers can be used with the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

12.4 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( )** function with the **FIOSETOPTIONS** function. For example, to set all the *tty* options except **OPT_MON_TRAP**:

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL & ~OPT_MON_TRAP);
```

For more information, see *Kernel I/O Control Functions*, p.325.

Table 12-2 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 12-2 **Tty Options**

Library	Description
OPT_LINE	Selects *line mode*. (See *Raw Mode and Line Mode*, p.324.)
OPT_ECHO	Echoes input characters to the output of the same channel.
OPT_CRMOD	Translates input **RETURN** characters into **NEWLINE** (\n); translates output **NEWLINE** into **RETURN-LINEFEED**.
OPT_TANDEM	Responds to software flow control characters **CTRL+Q** and **CTRL+S** (**XON** and **XOFF**).
OPT_7_BIT	Strips the most significant bit from all input bytes.
OPT_MON_TRAP	Enables the special *ROM monitor trap* character, **CTRL+X** by default.
OPT_ABORT	Enables the special kernel shell abort character, **CTRL+C** by default. (Only useful if the kernel shell is configured into the system)
OPT_TERMINAL	Sets all of the above option bits.
OPT_RAW	Sets none of the above option bits.

Raw Mode and Line Mode

A *tty* device operates in one of two modes: *raw mode* (unbuffered) or *line mode*. Raw mode is the default. Line mode is selected by the **OPT_LINE** bit of the device option word (see *tty Options*, p.323).

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.324.

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. (It is not possible to restart VxWorks if un-handled external interrupts occur during the boot countdown.) 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 function, 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** functions shown in Table 12-3.

Table 12-3 **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

Kernel I/O Control Functions

The *tty* devices respond to the **ioctl( )** functions in Table 12-4, defined in **ioLib.h**. For more information, see the reference entries for **tyLib**, **ttyDrv**, and **ioctl( )**.

Table 12-4 **I/O Control Functions Supported by tyLib**

Function	Description
FIOBAUDRATE	Sets the baud rate to the specified argument.
FIOCANCEL	Cancels a read or write.
FIODATASYNC	Discards all bytes in the input and output buffers.
FIOFLUSH	Same as **FIODATASYNC**.
FIOGETNAME	Gets the filename of the file descriptor.
FIOGETOPTIONS	Returns the current device option word.
FIONREAD	Gets the number of unread bytes in the input buffer.
FIONWRITE	Gets the number of bytes in the output buffer.
FIOSETOPTIONS	Sets the device option word.
FIOSYNC	Same as **FIODATASYNC**.

⚠ **CAUTION:** To change the driver's hardware options (for example, the number of stop bits or parity bits), use the **ioctl( )** function **SIO_HW_OPTS_SET**. Because this command is not implemented in most drivers, you may need to add it to your BSP serial driver. The details of how to implement this command depend on your board's serial chip. The constants defined in the header file **sioLib.h** provide the POSIX definitions for setting the hardware options.

12.5 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. For more information, see *7.14 Pipes*, p. 136.

12.6 RAM Drives

VxWorks provides facilities for several types of RAM drives (also called RAM disks), which treat a block of memory as if it were hardware storage media. Some types of RAM drives can be created using volatile as well a non-volatile RAM.

memDrv

A **memDrv** drive allows memory to be treated as a flat file with no block device. It provides a high-level method of reading and writing bytes in absolute memory locations through I/O calls. It can also be used to implement a simple, essentially read-only file system (existing files can be rewritten within their existing sizes); directory searches and a limited set of IOCTL calls are supported. It cannot, however, be used to implement a file system.

Memory location and size are specified when the device is created. This feature is useful when data must be preserved between warm reboots, or when sharing data between CPUs.

For more information, see the **memDrv** API reference entry.

xbdRamDisk

An **xbdRamDisk** drive allows the use of a file system in RAM memory. The RAM drive can be used with the HRFS, dosFs, and rawFs file systems. It links into the file system monitor and event framework.

Note that this XBD-compatible RAM drive facility supersedes the legacy **ramDrv** facility (which also supports file systems). However, you cannot specify the location of an **xbdRamDisk** drive in memory, whereas you can specify the location of a **ramDrv** drive.

For more the API reference for **xbdRamDisk**, as well as implementation examples in the *VxWorks 7 File Systems Programmer's Guide*.

ramDrv

A **ramDrv** drive uses a simple block device in RAM addressable by the I/O system. It can be formatted directly with the dosFs file system, or wrapped in an XBD block device and formatted with the HRFS file system.

While **ramDrv** is a legacy technology superseded by **xbdRamDisk**, it does allow you to specify the location of the drive in memory, which is useful in allowing recovery of RAM drive contents after a warm reboot.

For more information, see the **ramDrv** API reference entry.

romfFsDrv

A **romFsDrv** drive can be created in RAM or ROM memory. It must be associated with the ROMFS file system, which is a small portable hierarchical read-only file system. ROMFS is particularly useful for incorporating read-only files into a small-footprint system.

For more information, see the **romfsDrv** and **romfsLib** API reference entries. For information about implementing a ROMFS with VxWorks project configuration (instead of programmatically), see the *VxWorks 7 File Systems Programmer's Guide*.

12.7 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 *9.3 POSIX PSE52 Support*, p. 180. For information about VRFS, see the *VxWorks 7 File Systems Programmer's Guide*.

12.8 Block Devices

A physical *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk. In VxWorks, the term *block* refers to the smallest addressable unit on the device. For most disk devices, a VxWorks block corresponds to a *sector*, although terminology varies.

Block devices in VxWorks have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, the I/O activity of block device drivers is mediated by the extended block device (XBD) facility and a file system. The XBD facility provides a standard interface for block device drivers on the one hand, and for file systems on the other.

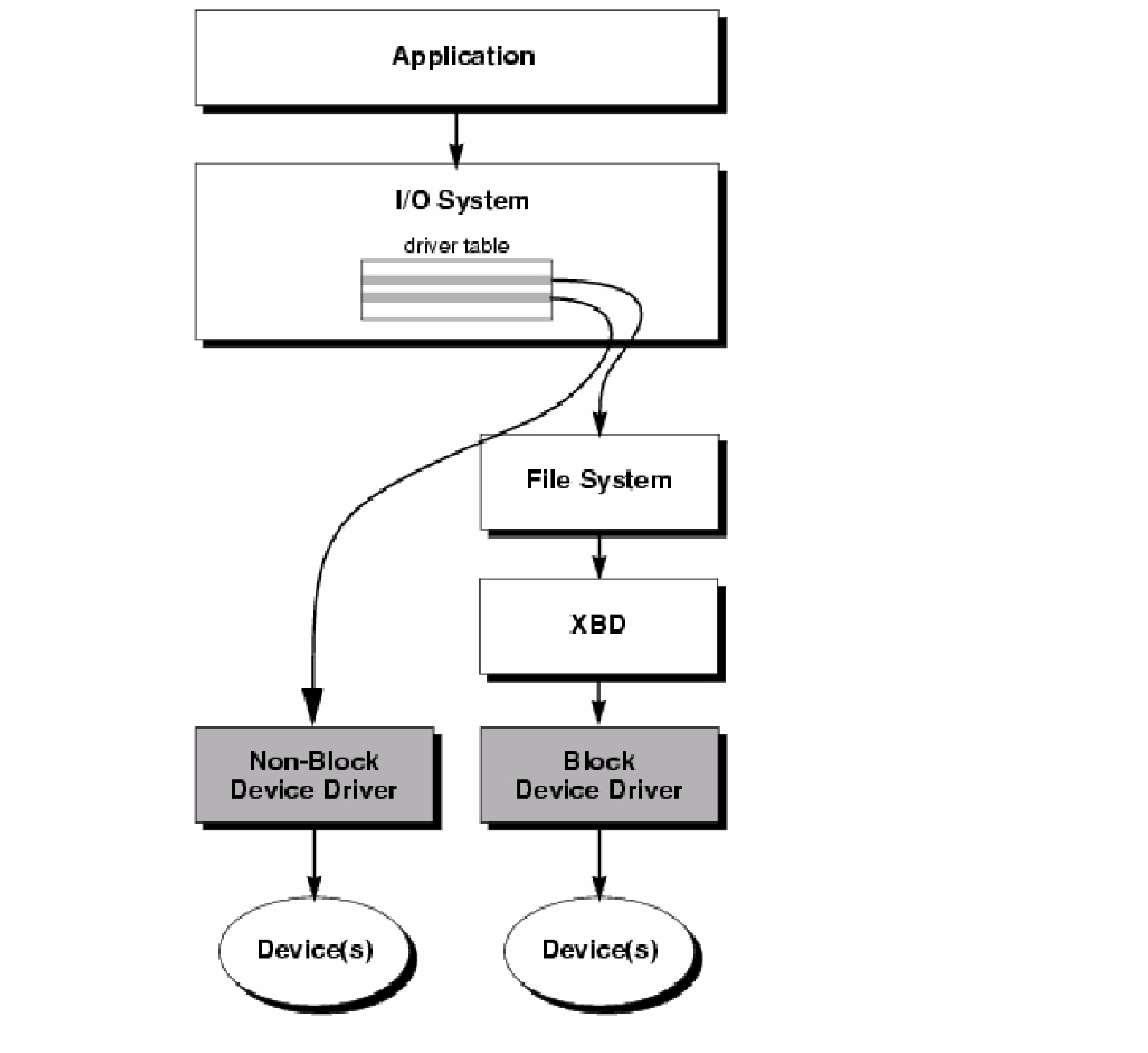
Figure 12-1 shows a layered model of I/O for both block and non-block (character) devices. This architecture allows the same block device driver to be used with different file systems, and reduces the number of I/O functions that must be supported in the driver.

For information about the XBD facility, see *12.9 Extended Block Device Facility: XBD*, p. 329.

For information about the file systems that can be used with block devices, see the *VxWorks 7 File Systems Programmer's Guide*.

For information about information about block device drivers and how to develop them, see the *VxWorks Device Driver Developer's Guide*.

Figure 12-1 **Non-Block Devices and Block Devices**



12.9 Extended Block Device Facility: XBD

The extended block device (XBD) facility mediates I/O activity between file systems and block devices. It provides a standard interface between file systems on the one hand, and block drivers on the other, dispatching I/O requests coming from a file system to the appropriate device driver.

The XBD facility also provides support for removable file systems, automatic file system detection, and multiple file systems. For more information on these features, see the *VxWorks 7 File Systems Programmer's Guide*.

NOTE: The XBD facility is required for some file systems (such as HRFS, dosFs, cdromFs, and rawFs), but not others (such as ROMFS).

For detailed information on developing XBD-compatible device drivers, see the *VxWorks Device Driver Developer's Guide*.

The basic XBD facility is proved with the **INCLUDE_XBD** component, which also provides generic service for following optional components:

INCLUDE_XBD_RAMDRV
Provides support for RAM disks. See *12.6 RAM Drives*, p.326.

INCLUDE_XBD_PART_LIB
Provides disk partitioning facilities. See *XBD Disk Partition Manager*, p.330.

INCLUDE_XBD_BLK_DEV
Provides support for legacy block device drivers that were designed to work with the predecessor to XBD—the cache block I/O (CBIO) facility. These devices include floppy drives, and TrueFFS (the disk-access emulator for flash). See *XBD Block Device Wrapper*, p.330.

XBD Disk Partition Manager

VxWorks provides support for PC-style disk partitioning with the **INCLUDE_XBD_PART_LIB** component, which facilitates sharing fixed disks and removable cartridges between VxWorks target systems and PCs running Windows. This component includes two modules: **xbdPartition**, which manages partitions; and **partLib**, which provides partitioning facilities.

The **xbdPartition** facility creates a device for each partition that is detected on media. Each partition that is detected is probed by the file system monitor and an I/O device is added to VxWorks. This device is an instantiation of the file system found by the file system monitor (or rawFs if the file system is not recognized or detected). If no partitions are detected, a single device is created to represent the entire media. There can be up to four partitions on a single media. (For information about the file system monitor, see the *VxWorks 7 File Systems Programmer's Guide*.)

The partition facility also names the partitions. The names are derived from the device name and the partition number (the default device name can be changed with the **INCLUDE_SATA_DISK_NAME_CFG** component).

For example, the name of an XBD-compatible device for an ATA hard disk without partitions would be **/ata0:0**.

If four partitions were created, they would be named as follows:

```
/ata0:1
/ata0:2
/ata0:3
/ata0:4
```

The **partLib** library provides facilities for creating PC-style partitions on media. It can create up to four primary partitions. When partitions are created, any existing information on the media is lost. For more information see the VxWorks API reference for **partLib**.

XBD Block Device Wrapper

The **INCLUDE_XBD_BLK_DEV** component provides support for legacy block devices that were designed to work the predecessor to XBD—the cache block I/O

(CBIO) facility. It provides a wrapper XBD facility that converts the block I/O driver interface based on the **BLK_DEV** logical block device structure into an XBD API-compliant interface.

NOTE: The Wind River devices that require the **INCLUDE_XBD_BLK_DEV** component in addition to **INCLUDE_XBD** are floppy, TrueFFS (the disk-access emulator for flash) drivers. Any third-party device drivers based on the **BLK_DEV** interface also require **INCLUDE_XBD_BLK_DEV**.

The Wind River drivers that do not require the **INCLUDE_XBD_BLK_DEV** component are USB block storage, ATA, and the XBD RAM disk.

CAUTION: Depending on the implementation of the driver, the **INCLUDE_XBD_BLK_DEV** component may not properly detect media insertion and removal. It may, therefore remove the file system when the media is removed, or not instantiate a file system when media is inserted.

12.10 PCMCIA

A PCMCIA card can be plugged into notebook computers to connect devices such as modems and external hard drives. VxWorks provides PCMCIA facilities for **Pentium** and **Pentium 4** BSPs and PCMCIA drivers that allow VxWorks running on these targets to support PCMCIA hardware.

PCMCIA support is at the PCMCIA Release 2.1 level. It does not include socket services or card services, which are not required by VxWorks. It does include chip drivers and libraries. The PCMCIA libraries and drivers are also available in source code form for VxWorks systems based on CPU architectures other than Intel Pentium.

To include PCMCIA support in your system, configure VxWorks with the **INCLUDE_PCMCIA** component. For information about PCMCIA facilities, see the API references for **pcmciaLib** and **pcmciaShow**.

12.11 Peripheral Component Interconnect: PCI

Peripheral Component Interconnect (PCI) is a bus standard for connecting peripherals to a PC, and is used in Pentium systems, among others. PCI includes buffers that de-couple the CPU from relatively slow peripherals, allowing them to operate asynchronously.

For information about PCI facilities, see the API references for **pciAutoConfigLib**, **pciConfigLib**, **pciInitLib**, and **pciConfigShow**.

12.12 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 using the VxWorks implementation of NFS, see the *VxWorks 7 File Systems Programmer's Guide*.

I/O Control Functions for NFS Clients

NFS client devices respond to the **ioctl( )** functions summarized in Table 12-5. The functions listed are defined in **ioLib.h**. For more information, see the reference entries for **nfsDrv**, **ioLib**, and **ioctl( )**.

Table 12-5 **Supported I/O Control Functions for Files Accessed through NFS**

IOCTL	Description
FIOGETNAME	Gets the file name of **fd** and copies it to the buffer referenced by **nameBuf**: `status = ioctl (fd, FIOGETNAME, &nameBuf);`
FIONREAD	Copies to **nBytesUnread** the number of bytes remaining in the file specified by **fd**: `status = ioctl (fd, FIONREAD, &nBytesUnread);`
FIOSEEK	Sets the current byte offset in the file to the position specified by **newOffset**. If the seek goes beyond the end-of-file, the file grows. The end-of-file pointer gets moved to the new position, and the new space is filled with zeros: `status = ioctl (fd, FIOSEEK, newOffset);`
FIOSYNC	Flush data to the remote NFS file. It takes no additional argument: `status = ioctl (fd, FIOSYNC, 0);`
FIOWHERE	Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument: `position = ioctl (fd, FIOWHERE, 0);`

Table 12-5 **Supported I/O Control Functions for Files Accessed through NFS** (cont'd)

IOCTL	Description

FIOREADDIR	Reads the next directory entry. Use the third argument in the **ioctl( )** call to supply a pointer to a directory descriptor of type **DIR**.
	`DIR dirStruct;`
	`fd = open ("directory", O_RDONLY);`
	`status = ioctl (fd, FIOREADDIR, &dirStruct);`
	Normally, you do not use the **FIOREADDIR** functionality directly. Instead, you would call **readdir( )**. See the reference entry for **dirLib**.

FIOFSTATGET	Gets file status information (directory entry data). Use the third argument in the **ioctl( )** call to supply a pointer to a **stat** structure that is filled with data describing the specified file. For example:
	`struct stat statStruct;`
	`fd = open ("file", O_RDONLY);`
	`status = ioctl (fd, FIOFSTATGET, &statStruct);`
	Normally, you do not use the **FIOFSTATGET** functionality directly. Instead, you would **stat( )** or **fstat( )** functions get file information. See the manual entry for **dirLib**.

FIOFSTATFSGET	Gets the file system parameters for and open file descriptor. Use the third argument in the **ioctl( )** call to supply a pointer to a **statfs** structure that is filled with data describing the underlying file system.
	`statfs statfsStruct;`
	`fd = open ("directory", O_RDONLY);`
	`status = ioctl (fd, FIOFSTATFSGET, &statfsStruct);`
	Normally, you do not use the **FIOFSTATFSGET** functionality directly. Instead, you would **stat( )** or **fstat( )** functions get file information. See the manual entry for **dirLib**.

12.13 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.

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.

Creating Network Devices

To access files on a remote host using either RSH or FTP, a network device must first be created by calling the kernel function **netDevCreate( )**. The arguments to **netDevCreate( )** are (1) the name of the device, (2) the name of the host the device accesses, and (3) which protocol to use: 0 (RSH) or 1 (FTP).

For example, the following call creates an RSH device called **mars:** that accesses the host **mars**. By convention, the name for a network device is the remote machine's name followed by a colon (**:**).

```
netDevCreate ("mars:", "mars", 0);
```

Files on a network device can be created, opened, and manipulated as if on a local disk. Thus, opening the file **mars:/usr/foo** actually opens **/usr/foo** on host **mars**.

Note that creating a network device allows access to any file or device on the remote system, while mounting an NFS file system allows access only to a specified file system.

For the files of a remote host to be accessible with RSH or FTP, permissions and user identification must be established on both the remote and local systems. Creating and configuring network devices is discussed in detail in *Wind River Network Stack Programmer's Guide: File Access Applications* and in the API reference entry for **netDrv**.

I/O Control Functions

RSH and FTP devices respond to the same **ioctl( )** functions as NFS devices except for **FIOSYNC** and **FIOREADDIR**. The functions are defined in the header file **ioLib.h**. For more information, see the API reference entries for **netDrv** and **ioctl( )**.

12.14 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 functions 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 functions are source-compatible with the BSD 4.4 UNIX socket functions and the Windows Sockets (Winsock 1.1) networking standard. Use of these functions is discussed in *Wind River Network Stack Programmer's Guide*.

12.15 Internal I/O System Structure

The VxWorks I/O system differs from most I/O systems in the way that the work of performing user I/O requests is distributed between the device-independent I/O system and the device drivers themselves.

In many systems, the device driver supplies a few functions to perform low-level I/O functions such as reading a sequence of bytes from, or writing them to, character-oriented devices. The higher-level protocols, such as communications protocols on character-oriented devices, are implemented in the device-independent part of the I/O system. The user requests are heavily processed by the I/O system before the driver functions get control.

While this approach is designed to make it easy to implement drivers and to ensure that devices behave as much alike as possible, it has several drawbacks. The driver writer is often seriously hampered in implementing alternative protocols that are not provided by the existing I/O system. In a real-time system, it is sometimes desirable to bypass the standard protocols altogether for certain devices where throughput is critical, or where the device does not fit the standard model.

In the VxWorks I/O system, minimal processing is done on user I/O requests before control is given to the device driver. The VxWorks I/O system acts as a switch to route user requests to appropriate driver-supplied functions. Each driver can then process the raw user requests as appropriate to its devices. In addition, however, several high-level libraries are available to driver writers that implement standard protocols for both character- and block-oriented devices. Thus the VxWorks I/O system provides the best of both worlds: while it is easy to write a standard driver for most devices with only a few pages of device-specific code, driver writers are free to execute the user requests in nonstandard ways where appropriate.

There are two fundamental types of device: *block* and *character* (or *non-block*; see Figure 12-1). Block devices are used for storing file systems. They are random access devices where data is transferred in blocks. Examples of block devices include hard and floppy disks. Character devices are typically of the tty/sio type.

As discussed in earlier sections, the three main elements of the VxWorks I/O system are drivers, devices, and files. The following sections describe these elements in detail. The discussion focuses on character drivers; however, much of it is applicable to block devices. Because block drivers must interact with VxWorks file systems, they use a slightly different organization.

NOTE: This discussion is designed to clarify the structure of VxWorks I/O facilities and to highlight some considerations relevant to writing I/O drivers for VxWorks. For detailed information about writing device drivers, see the *VxWorks Device Driver Developer's Guide*.

Example 12-1 shows the abbreviated code for a hypothetical driver that is used as an example throughout the following discussions. This example driver is typical of drivers for character-oriented devices.

In VxWorks, each driver has a short, unique abbreviation, such as **net** or **tty**, which is used as a prefix for each of its functions. The abbreviation for the example driver is *xx*.

Example 12-1 **Hypothetical Driver**

```
/*
 * xxDrv - driver initialization function
 * xxDrv() init's the driver. It installs the driver via iosDrvInstall.
 * It may allocate data structures, connect ISRs, and initialize hardware*/

STATUS xxDrv ()
  {
  xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl)
;
  (void) intConnect (intvec, xxInterrupt, ...);
  ...
  }

/*********************************************************************
 * xxDevCreate - device creation function
 *
 * Called to add a device called <name> to be svced by this driver. Other
 * driver-dependent arguments may include buffer sizes, device addresses.
 * The function adds the device to the I/O system by calling iosDevAdd.
 * It may also allocate and initialize data structures for the device,
 * initialize semaphores, initialize device hardware, and so on.
 */

STATUS xxDevCreate (name, ...)
  char * name;
  ...
  {
  status = iosDevAdd (xxDev, name, xxDrvNum);
  ...
  }

/*
 *
 * The following functions implement the basic I/O functions.
 * The xxOpen() return value is meaningful only to this driver,
 * and is passed back as an argument to the other I/O functions.
 */

int xxOpen (xxDev, remainder, mode)
  XXDEV * xxDev;
  char * remainder;
  int mode;
  {
  /* serial devices should have no file name part */

  if (remainder[0] != 0)
    return (ERROR);
  else
    return ((int) xxDev);
  }

int xxRead (xxDev, buffer, nBytes)
  XXDEV * xxDev;
  char * buffer;
  int nBytes;
  ...
int xxWrite (xxDev, buffer, nBytes)
  ...
int xxIoctl (xxDev, requestCode, arg)
  ...

/*
 * xxInterrupt - interrupt service function
 *
 * Most drivers have functions that handle interrupts from the devices
 * serviced by the driver. These functions are connected to the interrupts
 * by calling intConnect (usually in xxDrv above). They can receive a
```

```
  * single argument, specified in the call to intConnect (see intLib).
 */

 VOID xxInterrupt (arg)
   ...
```

Drivers

A driver for a non-block device generally implements the seven basic I/O functions—**creat( )**, **remove( )**, **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**—for a particular kind of device. The driver implements these general functions with corresponding device-specific functions that are installed with **iosDrvInstall( )**.

NOTE: Only VxBus-compatible drivers can be used with the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

Not all of the general I/O functions are implemented if they are not supported by a particular device. For example, **remove( )** is usually not supported for devices that are not used with file systems.

If any of the seven basic I/O functions are not implemented by a driver, a null function pointer should be used for the corresponding **iosDrvInstall( )** parameter when the driver is installed. Any call to a function that is not supported will then fail and return an **ENOTSUP** error.

Drivers may (optionally) allow tasks to wait for activity on multiple file descriptors. This functionality is implemented with the driver's **ioctl( )** function; see *Implementing select( )*, p.346.

A driver for a block device interfaces with a file system, rather than directly with the I/O system. The file system in turn implements most I/O functions. The driver need only supply functions to read and write blocks, reset the device, perform I/O control, and check device status.

When an application invokes one of the basic I/O functions, the I/O system routes the request to the appropriate function of a specific driver, as described in the following sections. The driver's function runs in the calling task's context, as though it were called directly from the application. Thus, the driver is free to use any facilities normally available to tasks, including I/O to other devices. This means that most drivers have to use some mechanism to provide mutual exclusion to critical regions of code. The usual mechanism is the semaphore facility provided in **semLib**.

In addition to the functions that implement the seven basic I/O functions, drivers also have three other functions:

- An initialization function that installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization. This function is typically named *xx***Drv( )**.
- A function to add devices that are to be serviced by the driver to the I/O system. This function is typically named *xx***DevCreate( )**.
- Interrupt-level functions that are connected to the interrupts of the devices serviced by the driver.

The Driver Table and Installing Drivers

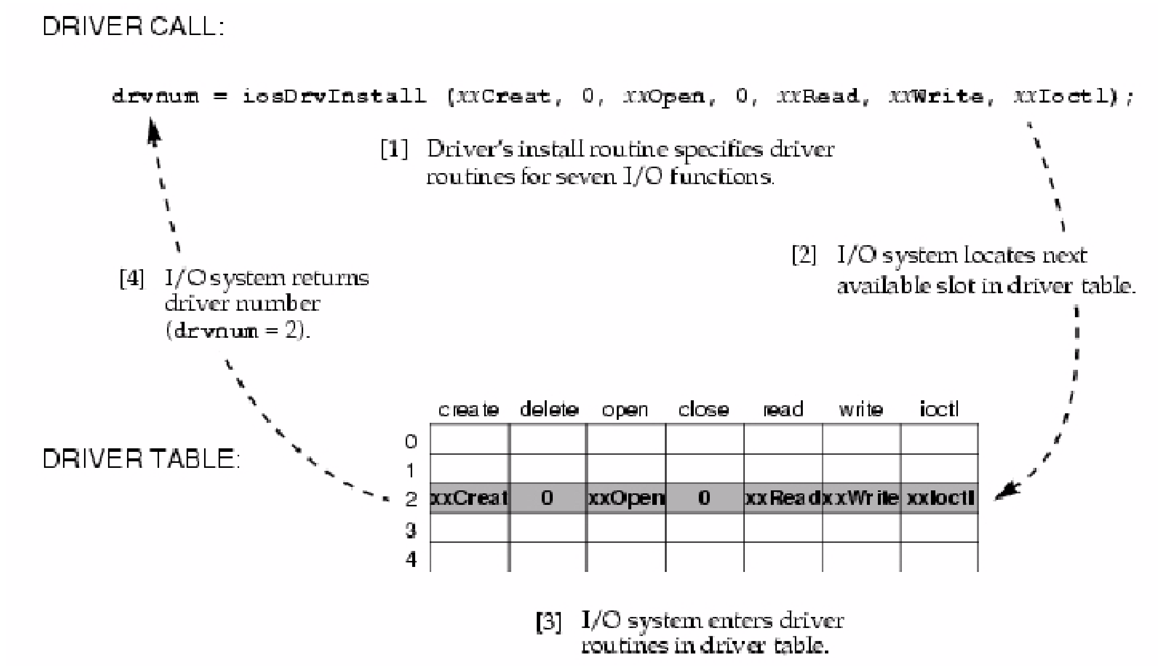
The function of the I/O system is to route user I/O requests to the appropriate function of the appropriate driver. The I/O system does this by maintaining a table that contains the address of each function for each driver. Drivers are installed dynamically by calling the I/O system internal function **iosDrvInstall( )**. The arguments to this function are the addresses of the seven I/O functions for the new driver. The **iosDrvInstall( )** function enters these addresses in a free slot in the driver table and returns the index of this slot. This index is known as the *driver number* and is used subsequently to associate particular devices with the driver.

Null (0) addresses can be specified for any of the seven basic I/O functions that are not supported by a device. For example, **remove( )** is usually not supported for non-file-system devices, and a null is specified for the driver's remove function.

When a user I/O call matches a null driver function, the call fails and an **ENOTSUP** error is returned.

VxWorks file systems (such as **dosFsLib**) contain their own entries in the driver table, which are created when the file system library is initialized.

Figure 12-2 **Example – Driver Initialization for Non-Block Devices**


DRIVER CALL:
drvnum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
[1] Driver's install routine specifies driver routines for seven I/O functions.
[2] I/O system locates next available slot in driver table.
[4] I/O system returns driver number (drvnum = 2).
DRIVER TABLE:
create delete open close read write ioctl
0
1
2 xxCreat 0 xxOpen 0 xxRead xxWrite xxIoctl
3
4
[3] I/O system enters driver routines in driver table.

Example of Installing a Driver

Figure 12-2 shows the actions taken by the example driver and by the I/O system when the initialization function *xx***Drv( )** runs.

The driver calls **iosDrvInstall( )**, specifying the addresses of the driver's functions for the seven basic I/O functions. Then, the I/O system:

1. Locates the next available slot in the driver table, in this case slot 2.
2. Enters the addresses of the driver functions in the driver table.
3. Returns the slot number as the driver number of the newly installed driver.

Devices

Some drivers are capable of servicing many instances of a particular kind of device. For example, a single driver for a serial communications device can often handle many separate channels that differ only in a few parameters, such as device address.

In the VxWorks I/O system, devices are defined by a data structure called a *device header* (**DEV_HDR**). This data structure contains the device name string and the driver number for the driver that services this device. The device headers for all the devices in the system are kept in a memory-resident linked list called the *device list*. The device header is the initial part of a larger structure determined by the individual drivers. This larger structure, called a *device descriptor*, contains additional device-specific data such as device addresses, buffers, and semaphores.

The Device List and Adding Devices

Non-block devices are added to the I/O system dynamically by calling the internal I/O function **iosDevAdd( )**. The arguments to **iosDevAdd( )** are the address of the device descriptor for the new device, the device's name, and the driver number of the driver that services the device. The device descriptor specified by the driver can contain any necessary device-dependent information, as long as it begins with a device header. The driver does not need to fill in the device header, only the device-dependent information. The **iosDevAdd( )** function enters the specified device name and the driver number in the device header and adds it to the system device list.

To add a block device to the I/O system, call the device initialization function for the file system required on that device. The device initialization function then calls **iosDevAdd( )** automatically.

The function **iosDevFind( )** can be used to locate the device structure (by obtaining a pointer to the **DEV_HDR**, which is the first member of that structure) and to verify that a device name exists in the table.

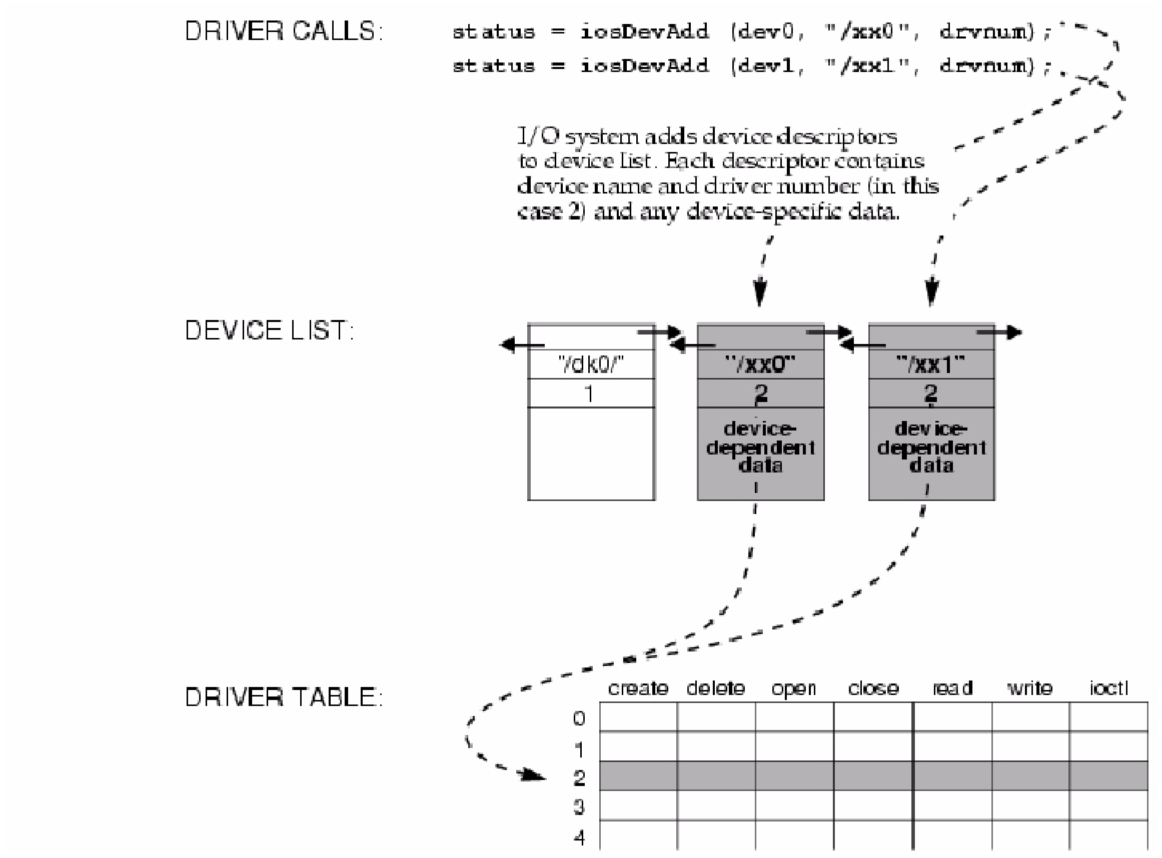
The following is an example using **iosDevFind( )**:

```
char *  pTail;                          /* pointer to tail of devName */
char devName[6] = "DEV1:";              /* name of device */
DOS_VOLUME_DESC *  pDosVolDesc;         /* first member is DEV_HDR */
    ...
    pDosVolDesc = iosDevFind(devName, (char**)&pTail);
    if (NULL == pDosVolDesc)
        {
        /* ERROR: device name does not exist and no default device */
        }
    else
        {
        /*
         * pDosVolDesc is a valid DEV_HDR pointer
         * and pTail points to beginning of devName.
         * Check devName against pTail to determine if it is
         * the default name or the specified devName.
         */
        }
```

Example of Adding Devices

In Figure 12-3, the example driver's device creation function *xx***DevCreate( )** adds devices to the I/O system by calling **iosDevAdd( )**.

Figure 12-3 **Example – Addition of Devices to I/O System**


DRIVER CALLS:
status = iosDevAdd (dev0, "/xx0", drvnum);
status = iosDevAdd (dev1, "/xx1", drvnum);
I/O system adds device descriptors to device list. Each descriptor contains device name and driver number (in this case 2) and any device-specific data.
DEVICE LIST:
"/dk0/"
1
"/xx0"
2
device-dependent data
"/xx1"
2
device-dependent data
DRIVER TABLE:
create
delete
open
close
read
write
ioctl
0
1
2
3
4

Deleting Devices

A device can be deleted with **iosDevDelete( )** and the associated driver removed with **iosDrvRemove( )**.

Note that a device-deletion operation causes the file descriptors that are open on the device to be *invalidated*, but not closed. The file descriptors can only be closed by an explicit act on the part of an application. If this were not the case, and file descriptors were closed automatically by the I/O system, the descriptors could be reassigned to new files while they were still being used by an application that was unaware of the deletion of the device. The new files could then be accessed unintentionally by an application attempting to use the files associated with the deleted device, as well as by an application that was correctly using the new files. This would result in I/O errors and possible device data corruption.

Because the file descriptors of a device that has been deleted are invalid, any subsequent I/O calls that use them—except **close( )**—will fail. The behavior of the I/O functions in this regard is as follows:

- **close( )** releases the file descriptor at I/O system level and the driver close function is not called.
- **read( )**, **write( )**, and **ioctl( )** fail with error **ENXIO** (no such device or address).
- While **open( )**, **remove( )**, and **create( )** do not take an open file descriptor as input, they fail because the device name is no longer in the device list.

Note that even if a device is deleted and immediately added again with the same device name, the file descriptors that were invalidated with the deletion are not restored to valid status. The behavior of the I/O calls on the associated file descriptors is the same as if the device had not been added again.

Applications that are likely to encounter device deletion should be sure to check for **ENXIO** errors from **read( )**, **write( )**, and **ioctl( )** calls, and to then close the relevant file descriptors.

Using Callback Functions to Manage Device Deletion

For situations in which devices are dynamically installed and deleted, the **iosDevDelCallback( )** function provides the means for calling a post-deletion handler after all driver invocations are exited.

A common use of a device deletion callback is to prevent a race condition that would result from a device descriptor being deleted in one thread of execution while it was still being used in another.

A device descriptor belongs to an application, and the I/O system cannot control its creation and release. It is a user data structure with **DEV_HDR** data structure embedded at the front of it, followed by any specific member of the device. Its pointer can be used to pass into any **iosDev***Xyz***( )** function as a **DEV_HDR** pointer, or used as the device descriptor for user device handling.

When a device is deleted, an application should not immediately release the device descriptor memory after **iosDevDelete( )** and **iosDrvRemove( )** calls because a driver invocation of the deleted device might still be in process. If the device descriptor is deleted while it is still in use by a driver function, serious errors could occur.

For example, the following would produce a race condition: task A invokes the driver function *xyz***Open( )** by a call to **open( )** and the *xyz***Open( )** call does not return before task B deletes the device and releases the device descriptor.

However, if descriptor release is not performed by task B, but by a callback function installed with **iosDevDelCallback( )**, then the release occurs only after task A's invocation of the driver function has finished.

A device callback function is called immediately when a device is deleted with **iosDevDelete( )** or **iosDrvRemove( )** as long as no invocations of the associated driver are operative (that is, the device driver reference counter is zero). Otherwise, the callback function is not executed until the last driver call exits (and the device driver reference counter reaches zero).

A device deletion callback function should be called with only one parameter, the pointer to the **DEV_HDR** data structure of the device in question. For example:

```
devDeleteCallback(pDevHdr)
```

The callback should be installed with **iosDevDelCallback( )** after the **iosDevAdd( )** call.

The following code fragments illustrate callback use. The file system device descriptor **pVolDesc** is installed into the I/O device list. Its device deletion callback, **fsVolDescRelease( )** performs the post-deletion processing, including releasing memory allocated for the device volume descriptor.

```
void fsVolDescRelease
   (
   FS_VOLUME_DESC * pVolDesc
   )
   {
   . . . . . .
   free (pVolDesc->pFsemList);
   free (pVolDesc->pFhdlList);
   free (pVolDesc->pFdList);
   . . . . . .
   }


STATUS fsDevCreate
   (
   char *  pDevName,   /* device name */
   device_t  device,   /* underlying block device */
   u_int   maxFiles,   /* max no. of simultaneously open files */
   u_int   devCreateOptions /* write option & volume integrity */
   )
   {
   FS_VOLUME_DESC  *pVolDesc = NULL; /* volume descriptor ptr */
   . . . . . .
   pVolDesc = (FS_VOLUME_DESC *) malloc (sizeof (*pVolDesc));
   pVolDesc->device = device;
   . . . . . .
   if (iosDevAdd((void *)pVolDesc, pDevName, fsDrvNum ) == ERROR)
      {
      pVolDesc->magic = NONE;
      goto error_iosadd;
      }
   /* Device deletion callback installed to release memory resource. */
   iosDevDelCallback((DEV_HDR *) pVolDesc, (FUNCPTR) fsVolDescRelease);
   . . . . . .
   }

STATUS fsDevDelete
   (
   FS_VOLUME_DESC *pVolDesc    /* pointer to volume descriptor */
   )
   {
   . . . . . .
   /*
    * Delete the file system device from I/O device list. Callback
    * fsVolDescRelease will be called from now on at a
    * safe time by I/O system.
    */
   iosDevDelete((DEV_HDR *) pVolDesc);
   . . . . . .
   }
```

The application should check the error returned by a deleted device, as follows:

```
if (write (fd, (char *)buffer, nbytes) == ERROR)
   {
   if (errno == ENXIO)
       {
       /* Device is deleted. fd must be closed by application. */
       close(fd);
       }
   else
       {
       /* write failure due to other reason. Do some error dealing. */
       . . . . . .
       }
       }
```

File Descriptors

Several file descriptors can be open to a single device at one time. A device driver can maintain additional information associated with a file descriptor beyond the I/O system’s device information. In particular, devices on which multiple files can be open at one time have file-specific information (for example, file offset) associated with each file descriptor. You can also have several file descriptors open to a non-block device, such as a *tty*; typically there is no additional information, and thus writing on any of the file descriptors produces identical results.

File Descriptor Table

Files are opened with **open( )** or **creat( )**. The I/O system searches the device list for a device name that matches the filename (or an initial substring) specified by the caller. If a match is found, the I/O system uses the driver number contained in the corresponding device header to locate and call the driver's open function in the driver table.

The I/O system must establish an association between the file descriptor used by the caller in subsequent I/O calls, and the driver that services it. Additionally, the driver must associate some data structure per descriptor. In the case of non-block devices, this is usually the device descriptor that was located by the I/O system.

The I/O system maintains these associations in a table called the *file descriptor table*. This table contains the driver number and an additional driver-determined 4-byte value. The driver value is the internal descriptor returned by the driver's open function, and can be any value the driver requires to identify the file. In subsequent calls to the driver's other I/O functions (**read( )**, **write( )**, **ioctl( )**, and **close( )**), this value is supplied to the driver in place of the file descriptor in the application-level I/O call.

Example of Opening a File

In Figure 12-4 and Figure 12-5, a user calls **open( )** to open the file **/***xx***0**. The I/O system takes the following series of actions:

1. It searches the device list for a device name that matches the specified filename (or an initial substring). In this case, a complete device name matches.

2. It reserves a slot in the file descriptor table and creates a new file descriptor object, which is used if the open is successful.

3. It then looks up the address of the driver's open function, *xx***Open( )**, and calls that function. Note that the arguments to *xx***Open( )** are transformed by the I/O system from the user's original arguments to **open( )**. The first argument to *xx***Open( )** is a pointer to the device descriptor the I/O system located in the full filename search. The next parameter is the *remainder* of the filename specified by the user, after removing the initial substring that matched the device name. In this case, because the device name matched the entire filename, the remainder passed to the driver is a null string. The driver is free to interpret this remainder in any way it wants. In the case of block devices, this remainder is the name of a file on the device. In the case of non-block devices like this one, it is usually an error for the remainder to be anything *but* the null string. The third parameter is the file access flag, in this case **O_RDONLY**; that is, the file is opened for reading only. The last parameter is the mode, as passed to the original **open( )** function.

4. It executes *xx***Open( )**, which returns a value that subsequently identifies the newly opened file. In this case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the file being opened. Note that if the driver returns only the device descriptor, the driver cannot distinguish multiple files opened to the same device. In the case of non-block device drivers, this is usually appropriate.

5. The I/O system then enters the driver number and the value returned by *xx***Open( )** in the new file descriptor object. Again, the value entered in the file descriptor object has meaning only for the driver, and is arbitrary as far as the I/O system is concerned.

6. Finally, it returns to the user the index of the slot in the file descriptor table, in this case 3.

Figure 12-4 **Example: Call to I/O Function open( ) [Part 1]**

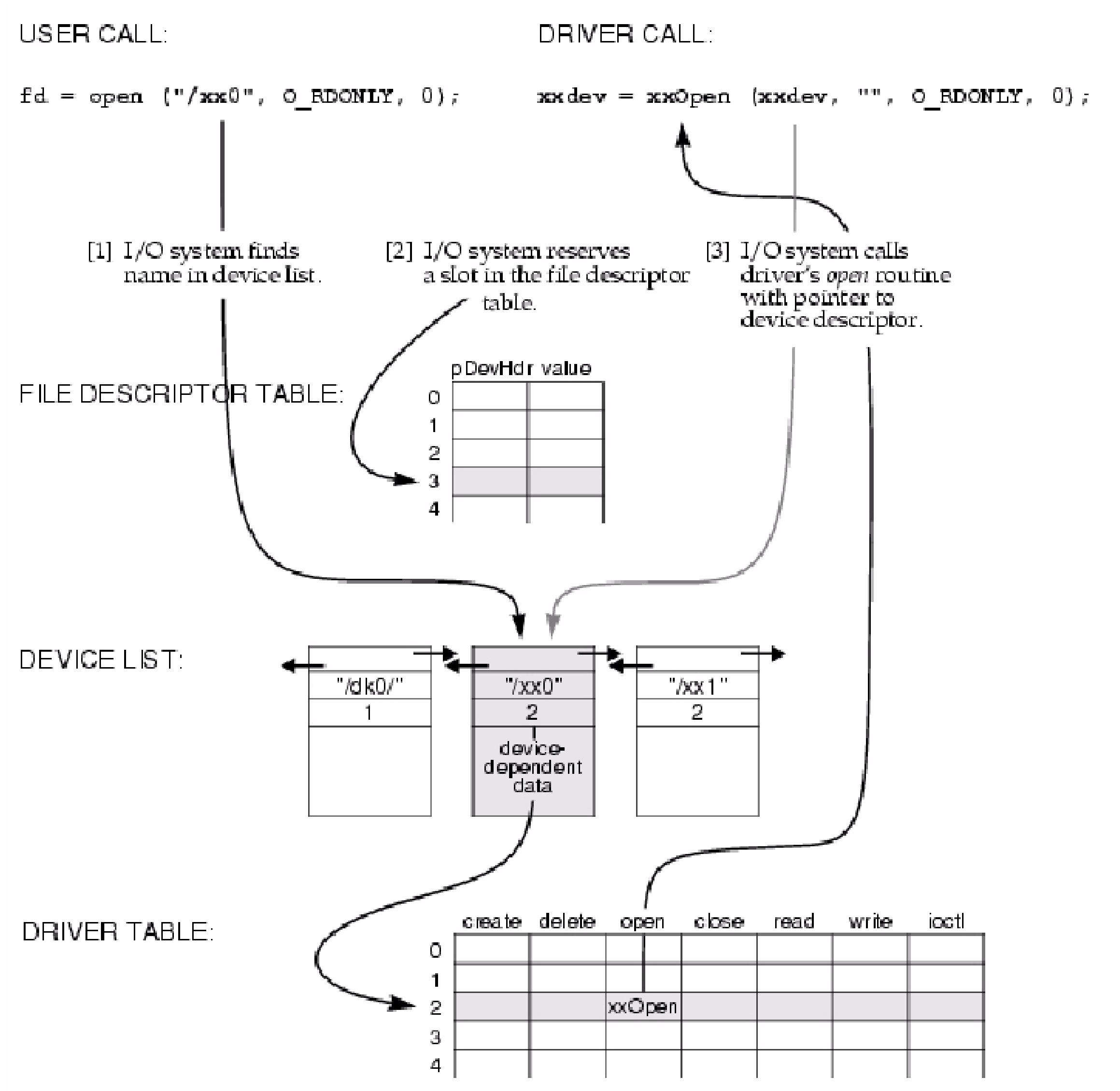
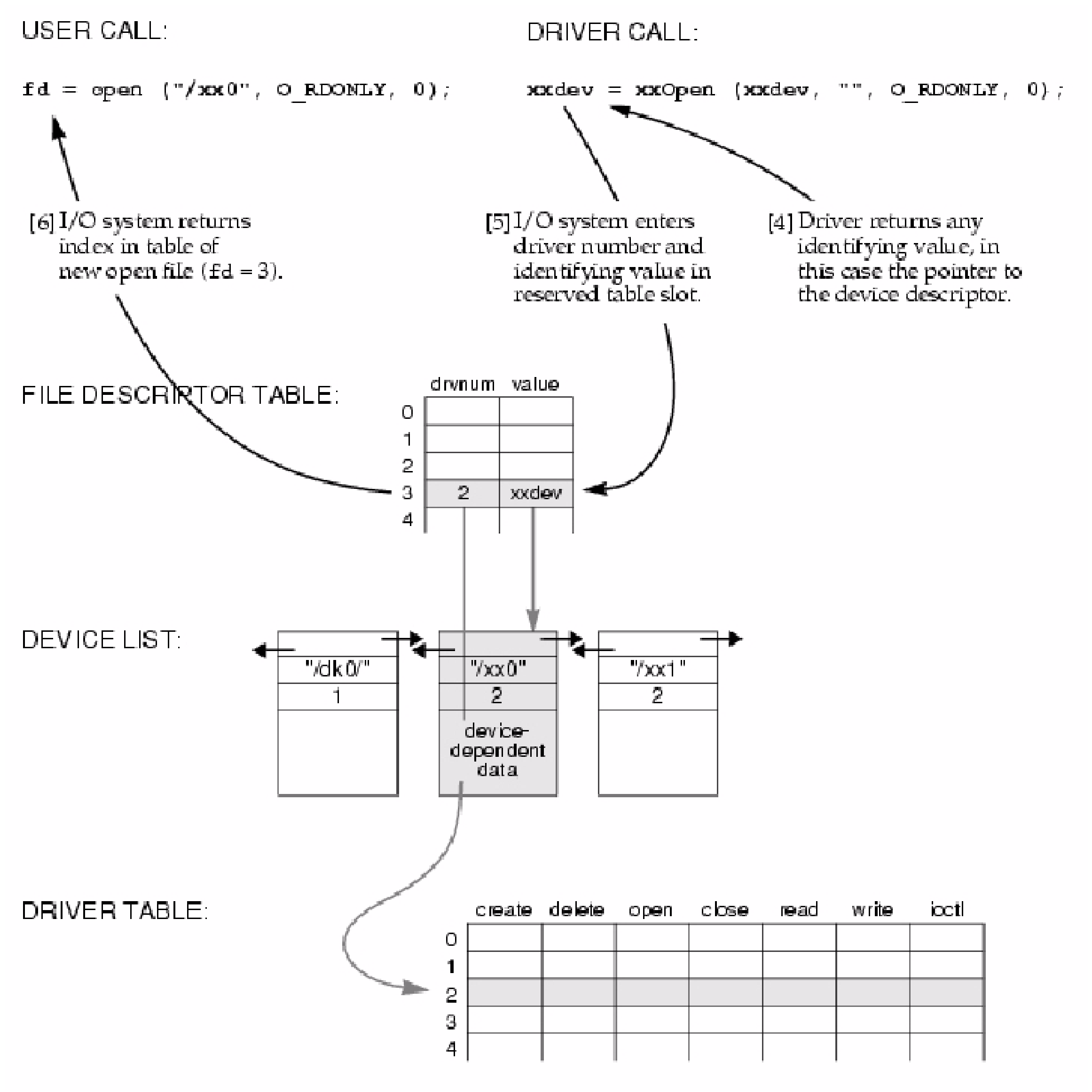

USER CALL:
fd = open ("/xx0", O_RDONLY, 0);
DRIVER CALL:
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
[1] I/O system finds name in device list.
[2] I/O system reserves a slot in the file descriptor table.
[3] I/O system calls driver's open routine with pointer to device descriptor.
FILE DESCRIPTOR TABLE:
pDevHdr value
0
1
2
3
4
DEVICE LIST:
"/dk0/"
1
"/xx0"
2
device-dependent data
"/xx1"
2
DRIVER TABLE:
create delete open close read write ioctl
0
1
2
3
4
xxOpen

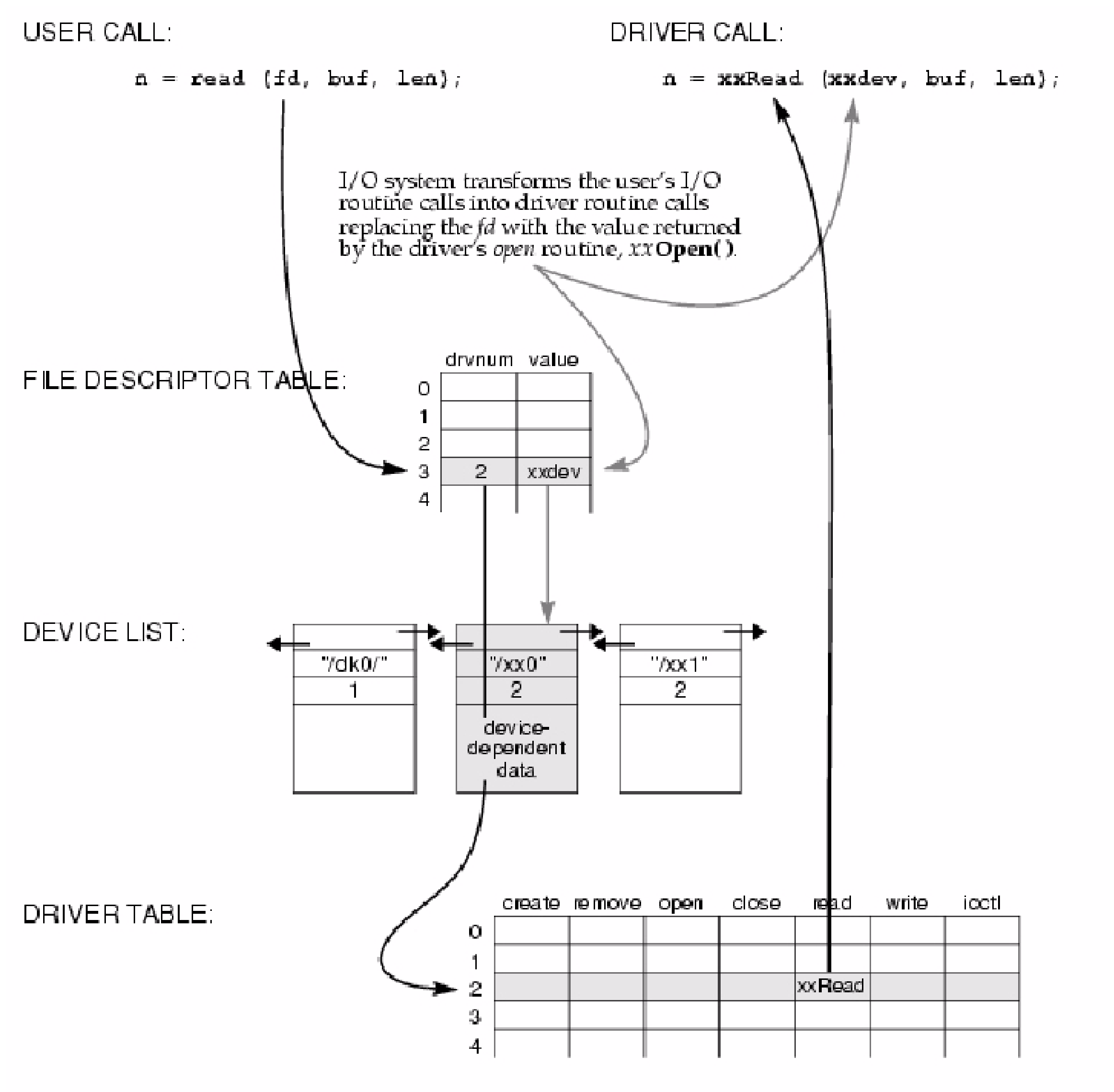
Figure 12-5 **Example: Call to I/O Function open( ) [Part 2]**


USER CALL:
fd = open ("/xx0", O_RDONLY, 0);
DRIVER CALL:
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
[6] I/O system returns index in table of new open file (fd = 3).
[5] I/O system enters driver number and identifying value in reserved table slot.
[4] Driver returns any identifying value, in this case the pointer to the device descriptor.
FILE DESCRIPTOR TABLE:
drvnum
value
0
1
2
3
2
xxdev
4
DEVICE LIST:
"/dk0/"
1
"/xx0"
2
device-dependent data
"/xx1"
2
DRIVER TABLE:
create
delete
open
close
read
write
ioctl
0
1
2
3
4

Example of Reading Data from the File

In Figure 12-6, the user calls **read( )** to obtain input data from the file. The specified file descriptor is the index into the file descriptor table for this file. The I/O system uses the driver number contained in the table to locate the driver's read function, *xx*__Read( )__. The I/O system calls *xx*__Read( )__, passing it the identifying value in the file descriptor table that was returned by the driver's open function, *xx*__Open( )__. Again, in this case the value is the pointer to the device descriptor. The driver's read function then does whatever is necessary to read data from the device. The process for user calls to **write( )** and **ioctl( )** follow the same procedure.

Figure 12-6 **Example: Call to I/O Function read( )**



Example of Closing a File

The user terminates the use of a file by calling **close( )**. As in the case of **read( )**, the I/O system uses the driver number contained in the file descriptor table to locate the driver's close function. In the example driver, no close function is specified; thus no driver functions are called. Instead, the I/O system marks the slot in the file descriptor table as being available. Any subsequent references to that file descriptor cause an error. Subsequent calls to **open( )** can reuse that slot.

Implementing select( )

Supporting **select( )** in your driver allows tasks to wait for input from multiple devices or to specify a maximum time to wait for the device to become ready for I/O. Writing a driver that supports **select( )** is simple, because most of the functionality is provided in **selectLib**. You might want your driver to support **select( )** if any of the following is appropriate for the device:

- The tasks want to specify a timeout to wait for I/O from the device. For example, a task might want to time out on a UDP socket if the packet never arrives.
- The driver supports multiple devices, and the tasks want to wait simultaneously for any number of them. For example, multiple pipes might be used for different data priorities.
- The tasks want to wait for I/O from the device while also waiting for I/O from another device. For example, a server task might use both pipes and sockets.

To implement **select( )**, the driver must keep a list of tasks waiting for device activity. When the device becomes ready, the driver unblocks all the tasks waiting on the device.

For a device driver to support **select( )**, it must declare a **SEL_WAKEUP_LIST** structure (typically declared as part of the device descriptor structure) and initialize it by calling **selWakeupListInit( )**. This is done in the driver's *xx***DevCreate( )** function. When a task calls **select( )**, **selectLib** calls the driver's **ioctl( )** function with the function **FIOSELECT** or **FIOUNSELECT**. If **ioctl( )** is called with **FIOSELECT,** the driver must do the following:

1. Add the **SEL_WAKEUP_NODE** (provided as the third argument of **ioctl( )**) to the **SEL_WAKEUP_LIST** by calling **selNodeAdd( )**.
2. Use the function **selWakeupType( )** to check whether the task is waiting for data to read from the device (**SELREAD**) or if the device is ready to be written (**SELWRITE**).
3. If the device is ready (for reading or writing as determined by **selWakeupType( )**), the driver calls the function **selWakeup( )** to make sure that the **select( )** call in the task does not pend. This avoids the situation where the task is blocked but the device is ready.

If **ioctl( )** is called with **FIOUNSELECT**, the driver calls **selNodeDelete( )** to remove the provided **SEL_WAKEUP_NODE** from the wakeup list.

When the device becomes available, **selWakeupAll( )** is used to unblock all the tasks waiting on this device. Although this typically occurs in the driver's ISR, it can also occur elsewhere. For example, a pipe driver might call **selWakeupAll( )** from its *xx***Read( )** function to unblock all the tasks waiting to write, now that there is room in the pipe to store the data. Similarly the pipe's *xx***Write( )** function might call **selWakeupAll( )** to unblock all the tasks waiting to read, now that there is data in the pipe.

Example 12-2 **Driver Code Using the Select Facility**

```
/* This code fragment shows how a driver might support select(). In this
 * example, the driver unblocks tasks waiting for the device to become ready
 * in its interrupt service function.
 */

/* myDrvLib.h - header file for driver */

typedef struct     /* MY_DEV */
    {
    DEV_HDR     devHdr;                  /* device header */
    BOOL       myDrvDataAvailable;       /* data is available to read */
    BOOL       myDrvRdyForWriting;       /* device is ready to write */
    SEL_WAKEUP_LIST selWakeupList;       /* list of tasks pended in select */
    } MY_DEV;
```

```
/* myDrv.c - code fragments for supporting select() in a driver */

#include <vxWorks.h>
#include <selectLib.h>

/* First create and initialize the device */


STATUS myDrvDevCreate
    (
    char *  name,                       /* name of device to create */
    )

    {
    MY_DEV * pMyDrvDev;                 /* pointer to device descriptor*/
    ... additional driver code ...

    /* allocate memory for MY_DEV */
    pMyDrvDev = (MY_DEV *) malloc (sizeof MY_DEV);
    ... additional driver code ...

    /* initialize MY_DEV */
    pMyDrvDev->myDrvDataAvailable=FALSE
    pMyDrvDev->myDrvRdyForWriting=FALSE

    /* initialize wakeup list */
    selWakeupListInit (&pMyDrvDev->selWakeupList);
    ... additional driver code ...
    }

/* ioctl function to request reading or writing */

STATUS myDrvIoctl
    (
    MY_DEV * pMyDrvDev,                /* pointer to device descriptor */
    int      request,                  /* ioctl function */
    int      arg                       /* where to send answer */
    )
    {
    ... additional driver code ...

    switch (request)
        {
        ... additional driver code ...

        case FIOSELECT:

            /* add node to wakeup list */

            selNodeAdd (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELREAD
                && pMyDrvDev->myDrvDataAvailable)
                {
                /* data available, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELWRITE
                && pMyDrvDev->myDrvRdyForWriting)
                {
                /* device ready for writing, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            break;
            case FIOUNSELECT:

                /* delete node from wakeup list */
                selNodeDelete (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);
                break;

                ... additional driver code ...
            }
        }

/* code that actually uses the select() function to read or write */

void myDrvIsr
    (
    MY_DEV * pMyDrvDev;
    )
    {
    ... additional driver code ...

    /* if there is data available to read, wake up all pending tasks */

    if (pMyDrvDev->myDrvDataAvailable)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELREAD);

    /* if the device is ready to write, wake up all pending tasks */

    if (pMyDrvDev->myDrvRdyForWriting)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELWRITE);
    }
```

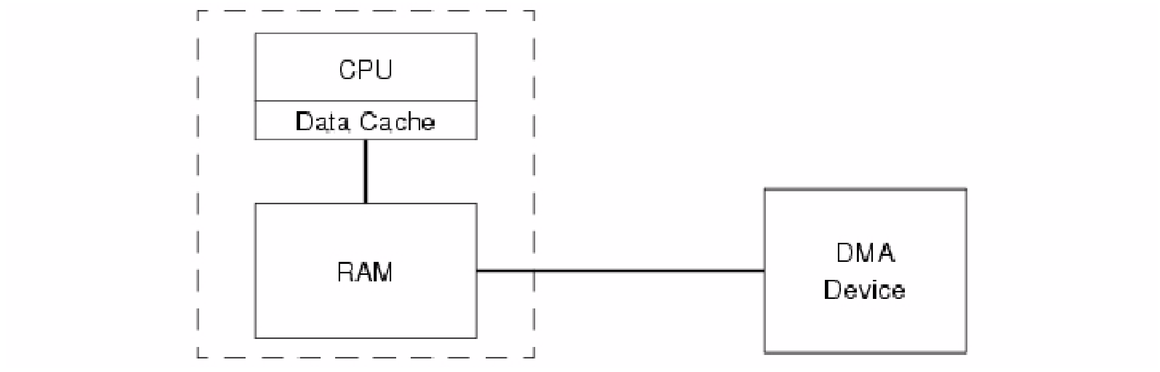
Cache Coherency

NOTE: The kernel cache facilities described in this section are provided for the uniprocessor (UP) configuration of VxWorks, some of which are not appropriate—and not provided—for the symmetric multiprocessor (SMP) configuration. For more information in this regard, see *cacheLib Restrictions*, p.453. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

Drivers written for boards with caches must guarantee *cache coherency*. Cache coherency means data in the cache must be in sync, or coherent, with data in RAM. The data cache and RAM can get out of sync any time there is asynchronous access to RAM (for example, DMA device access or VMEbus access). Data caches are used to increase performance by reducing the number of memory accesses. Figure 12-7 shows the relationships between the CPU, data cache, RAM, and a DMA device.

Data caches can operate in one of two modes: *writethrough* and *copyback*. Write-through mode writes data to both the cache and RAM; this guarantees cache coherency on output but not input. Copyback mode writes the data only to the cache; this makes cache coherency an issue for both input and output of data.

Figure 12-7 **Cache Coherency**



If a CPU writes data to RAM that is destined for a DMA device, the data can first be written to the data cache. When the DMA device transfers the data from RAM, there is no guarantee that the data in RAM was updated with the data in the cache. Thus, the data output to the device may not be the most recent—the new data may still be sitting in the cache. This data incoherence can be solved by making sure the data cache is flushed to RAM before the data is transferred to the DMA device.

If a CPU reads data from RAM that originated from a DMA device, the data read can be from the cache buffer (if the cache buffer for this data is not marked invalid) and not the data just transferred from the device to RAM. The solution to this data incoherence is to make sure that the cache buffer is marked invalid so that the data is read from RAM and not from the cache.

Drivers can solve the cache coherency problem either by allocating cache-safe buffers (buffers that are marked non-cacheable) or flushing and invalidating cache entries any time the data is written to or read from the device. Allocating cache-safe buffers is useful for static buffers; however, this typically requires MMU support. Non-cacheable buffers that are allocated and freed frequently (dynamic buffers) can result in large amounts of memory being marked non-cacheable. An alternative to using non-cacheable buffers is to flush and invalidate cache entries manually; this allows dynamic buffers to be kept coherent.

The functions **cacheFlush( )** and **cacheInvalidate( )** are used to manually flush and invalidate cache buffers. Before a device reads the data, flush the data from the cache to RAM using **cacheFlush( )** to ensure the device reads current data. After the device has written the data into RAM, invalidate the cache entry with **cacheInvalidate( )**. This guarantees that when the data is read by the CPU, the cache is updated with the new data in RAM.

Example 12-3 **DMA Transfer Function**

```
/* This a sample DMA transfer function. Before programming the device
 * to output the data to the device, it flushes the cache by calling
 * cacheFlush(). On a read, after the device has transferred the data,
 * the cache entry must be invalidated using cacheInvalidate().
 */

#include <vxWorks.h>
#include <cacheLib.h>
#include <fcntl.h>
#include "example.h"
void exampleDmaTransfer   /* 1 = READ, 0 = WRITE */
    (
    UINT8 *pExampleBuf,
    int exampleBufLen,
    int xferDirection
    )
    {
    if (xferDirection == 1)
        {
        myDevToBuf (pExampleBuf);
        cacheInvalidate (DATA_CACHE, pExampleBuf, exampleBufLen);
        }

    else
        {
        cacheFlush (DATA_CACHE, pExampleBuf, exampleBufLen);
        myBufToDev (pExampleBuf);
        }
    }
```

It is possible to make a driver more efficient by combining cache-safe buffer allocation and cache-entry flushing or invalidation. The idea is to flush or invalidate a cache entry only when absolutely necessary. To address issues of cache coherency for static buffers, use **cacheDmaMalloc( )**. This function initializes a **CACHE_FUNCS** structure (defined in **cacheLib.h**) to point to flush and invalidate functions that can be used to keep the cache coherent.

The macros **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE** use this structure to optimize the calling of the flush and invalidate functions. If the corresponding function pointer in the **CACHE_FUNCS** structure is NULL, no unnecessary flush/invalidate functions are called because it is assumed that the buffer is cache coherent (hence it is not necessary to flush/invalidate the cache entry manually).

The driver code uses a virtual address and the device uses a physical address. Whenever a device is given an address, it must be a physical address. Whenever the driver accesses the memory, it must use the virtual address.

The device driver should use **CACHE_DMA_VIRT_TO_PHYS** to translate a virtual address to a physical address before passing it to the device. It may also use **CACHE_DMA_PHYS_TO_VIRT** to translate a physical address to a virtual one, but this process is time-consuming and non-deterministic, and should be avoided whenever possible.

Example 12-4 **Address-Translation Driver**

```
/* The following code is an example of a driver that performs address
 * translations. It attempts to allocate a cache-safe buffer, fill it, and
 * then write it out to the device. It uses CACHE_DMA_FLUSH to make sure
 * the data is current. The driver then reads in new data and uses
 * CACHE_DMA_INVALIDATE to guarantee cache coherency.
 */

#include <vxWorks.h>
#include <cacheLib.h>
#include "myExample.h"
STATUS myDmaExample (void)
    {
    void * pMyBuf;
    void * pPhysAddr;

    /* allocate cache safe buffers if possible */
    if ((pMyBuf = cacheDmaMalloc (MY_BUF_SIZE)) == NULL)
    return (ERROR);

    … fill buffer with useful information …

    /* flush cache entry before data is written to device */
    CACHE_DMA_FLUSH (pMyBuf, MY_BUF_SIZE);
    /* convert virtual address to physical */
    pPhysAddr = CACHE_DMA_VIRT_TO_PHYS (pMyBuf);

    /* program device to read data from RAM */
    myBufToDev (pPhysAddr);
    … wait for DMA to complete …
    … ready to read new data …

    /* program device to write data to RAM */
    myDevToBuf (pPhysAddr);
    … wait for transfer to complete …

    /* convert physical to virtual address */
    pMyBuf = CACHE_DMA_PHYS_TO_VIRT (pPhysAddr);

    /* invalidate buffer */
    CACHE_DMA_INVALIDATE (pMyBuf, MY_BUF_SIZE);
    … use data ...

    /* when done free memory */
    if (cacheDmaFree (pMyBuf) == ERROR)
        return (ERROR);
    return (OK);
    }
```

13 Error Detection and Reporting

13.1 About Error Detection and Reporting Facilities

The VxWorks error detection and reporting facility facilitates 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:

- 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.
- A persistent memory region in RAM used to retain error records across warm reboots.
- The ability to display error records and clear the error log from the shell.
- Support for custom error-handing policies for responding to fatal errors.
- Macros for implementing error reporting in user code.

The hook management functions described in the **edrLib** entry in the *VxWorks Kernel API Reference* can be used as the basis for implementing custom functionality for non-RAM storage for error records.

For more information about error detection and reporting functions in addition to that provided in this chapter, see the *VxWorks Kernel API Reference* entries for **edrLib**, **edrShow**, **edrErrLogLib**, and **edrSysDbgLib**. For user mode, see the *VxWorks Application API Reference* entries for **edrLib**.

13.2 Configuration for Error Detection and Reporting

To use the error detection and reporting facilities you must select the appropriate components, allocate memory, and configure the facility itself.

More precisely, this means that you must:

- Configure VxWorks with support for error detection and reporting.
- Configure a persistent RAM memory region, which must be sufficiently large to hold the error records.
- Select debug or deployed mode, which specify different responses to fatal errors.
- Optionally, implement custom fatal error handling policies.

Configuring VxWorks

No special configuration of your VSB project is required. Support for error detection and reporting is provided in the core VxWorks code.

Add the following components to your VIP project:

- **INCLUDE_EDR_PM**
- **INCLUDE_EDR_ERRLOG**
- **INCLUDE_EDR_SHOW**
- **INCLUDE_EDR_SYSDBG_FLAG**

As a convenience, the **BUNDLE_EDR** component bundle may be used to include all of the above components.

Configuring the Persistent Memory Region

The persistent-memory region is an area of RAM 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.

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 *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

A cold reboot always clears the persistent memory region. The **pmInvalidate( )** function 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.

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 *Reserved Memory Configuration: User-Reserved Memory and Persistent Memory*, p.250 and the **pmLib** entry in the *VxWorks Kernel API Reference*.

Configuring Responses to Fatal Errors

The error detection and reporting facilities provide for two sets of responses to fatal errors. See *13.5 About Fatal Error Response Modes*, p.358 for information about these responses, and various ways to select one for a run-time system.

13.3 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 *13.5 About Fatal Error Response Modes*, p.358).

The event types are defined in Table 13-1, and the severity levels in Table 13-2.

Table 13-1 **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 13-2 **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

For an example of an error record, see *13.9 Sample Error Record*, p.360.

13.4 APIs for 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.

Table 13-3 **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.

CAUTION: Error records produced by 32-bit VxWorks cannot be read by 64-bit VxWorks and vice versa. Moreover, the persistent memory region of one version may be corrupted when the other version boots, and any existing records destroyed.

13.5 About Fatal Error Response Modes

In addition to generating error records, the error detection and reporting facility provides for two modes of system response to fatal errors (which have the **FATAL** severity level) for each event type.

The modes are:

- debug mode, for lab systems (development)
- deployed mode, for production systems (field)

NOTE: The operative error handling mode can be changed with system debug flag (see *13.6 Fatal Error Handling Mode*, p.359). The default is deployed mode.

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 difference between debug and deployed mode 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.

Table 13-4 describes the responses in each mode for each of the event types. It also lists the functions that are called when fatal records are created.

Table 13-4 **FATAL Error-Handling Options**

Event Type	Debug Mode	Deployed Mode (default)	Error Handling Function
INIT	Reboot	Reboot	**edrInitFatalPolicyHandler( )**
KERNEL	Stop failed task	Stop failed task	**edrKernelFatalPolicyHandler( )**
INTERRUPT	Reboot	Reboot	**edrInterruptFatalPolicyHandler( )**
RTP	Stop process	Delete process	**edrRtpFatalPolicyHandler( )**

The error handling functions are called in response to fatal errors—only errors with the **FATAL** severity level have handlers associated with them.

These handlers are defined in *installDir***/vxworks-7/pkgs/os/core/kernel-***x.x.x.x***/configlette/edrStub.c**. Developers can modify the functions in this file to implement different system responses to fatal errors. The names of the functions, however, cannot be changed.

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.

13.6 Fatal Error Handling Mode

Fatal error handling mode can be set for the error detection and reporting facility when VxWorks is configured with components, or when it is booted, or at runtime.

When VxWorks is configured with the **INCLUDE_EDR_SYSDBG_FLAG** component, the error detection and reporting facility responds to fatal errors in *debug mode*. When the component is not included, it responds in *deployed mode*.

If the **INCLUDE_EDR_SYSDBG_FLAG** component is included, the facility can also be set or reset to either mode at boot time or runtime:

- At boot time the mode can be set with the VxWorks boot line **flags** parameter. The value of 0x000 specifies deployed mode. The value of 0x400 (**SYSFLG_SYS_MODE_DEBUG**) specifies debug mode.
- At runtime the mode can be set programmatically. Use the **sysDebugModeGet( )** and **sysDebugModeSet( )** kernel functions to get the current status of the system debug flag and to set it.

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 mode.

13.7 Other Error Handling Options for RTPs

By default, any faults generated by an RTP (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 *8Signals, ISRs, and Watchdog Timers*, p. 151.

13.8 Use of 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.

For kernel applications see **edrLib.h** in *installDir***/vxworks-7/pkgs/os/core/kernel-***x.x.x.x***/h**.

For RTP applications, see **edrLib.h** is in *installDir***/vxworks-7/pkgs/os/core/user-***x.x.x.x***/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.

13.9 Sample Error Record

These sample error records for kernel task and RTP task failures illustrate the data provided by the error detection and reporting facility.

Kernel Task Failure

The following is an example of a record generated by a failed kernel task:

```
==[1/1]==============================================================
Severity/Facility:   FATAL/KERNEL
Boot Cycle:          1
```

```
OS Version:          6.0.0
Time:                THU JAN 01 05:15:07 1970 (ticks = 1134446)
Task:                "kernelTask" (0x0068c6c8)
Injection Point:     excArchLib.c:2523

fatal kernel task-level exception!

<<<<<Memory Map>>>>>

0x00100000 -> 0x002a48dc: kernel

<<<<<Exception Information>>>>>

data access
Exception current instruction address: 0x002110cc
Machine Status Register: 0x0000b032
Data Access Register: 0x50000000
Condition Register: 0x20000080
Data storage interrupt Register: 0x40000000

<<<<<Registers>>>>>

r0         = 0x00210ff8   sp         = 0x006e0f50   r2         = 0x00000000
r3         = 0x00213a10   r4         = 0x00003032   r5         = 0x00000001
r6         = 0x0068c6c8   r7         = 0x0000003a   r8         = 0x00000000
r9         = 0x00000000   r10        = 0x00000002   r11        = 0x00000002
r12        = 0x0000007f   r13        = 0x00000000   r14        = 0x00000000
r15        = 0x00000000   r16        = 0x00000000   r17        = 0x00000000
r18        = 0x00000000   r19        = 0x00000000   r20        = 0x00000000
r21        = 0x00000000   r22        = 0x00000000   r23        = 0x00000000
r24        = 0x00000000   r25        = 0x00000000   r26        = 0x00000000
r27        = 0x00000000   r28        = 0x00000000   r29        = 0x006e0f74
r30        = 0x00000000   r31        = 0x50000000   msr        = 0x0000b032
lr         = 0x00210ff8   ctr        = 0x0024046c   pc         = 0x002110cc
cr         = 0x20000080   xer        = 0x20000000   pgTblPtr   = 0x00481000
scSrTblPtr = 0x0047fe4c   srTblPtr   = 0x0047fe4c

<<<<<Disassembly>>>>>

 0x2110ac  2c0b0004    cmpi        crf0,0,r11,0x4 # 4
 0x2110b0  41820024    bc          0xc,2, 0x2110d4 # 0x002110d4
 0x2110b4  2c0b0008    cmpi        crf0,0,r11,0x8 # 8
 0x2110b8  41820030    bc          0xc,2, 0x2110e8 # 0x002110e8
 0x2110bc  4800004c    b           0x211108 # 0x00211108
 0x2110c0  3c600021    lis         r3,0x21 # 33
 0x2110c4  83e1001c    lwz         r31,28(r1)
 0x2110c8  38633a10    addi        r3,r3,0x3a10 # 14864
*0x2110cc  a09f0000    lhz         r4,0(r31)
 0x2110d0  48000048    b           0x211118 # 0x00211118
 0x2110d4  83e1001c    lwz         r31,28(r1)
 0x2110d8  3c600021    lis         r3,0x21 # 33
 0x2110dc  38633a15    addi        r3,r3,0x3a15 # 14869
 0x2110e0  809f0000    lwz         r4,0(r31)
 0x2110e4  48000034    b           0x211118 # 0x00211118
 0x2110e8  83e1001c    lwz         r31,28(r1)

<<<<<Traceback>>>>>

0x0011047c vxTaskEntry  +0x54 : 0x00211244 ()
0x00211258 d            +0x18 : memoryDump ()
```

RTP Task Failure

```
==[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 ()
```

14
RTP Core Dumps

14.1 About RTP Core Dumps

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.

An 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 generated 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 enable or disable 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 *14.7 Generating RTP Core Dump Generation*, p.368.

Data

An RTP core dump includes the following data:

- The RTP's memory space.
- The RTP's tasks.
- The data segment of 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 *14.10 RTP Core Dump Analysis*, p.369.

Size

The total size of the RTP core dump is determined as follows:

RTP data + shared libraries data + shared data

Filtering

The RTP core dump facility provides a function 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 *14.4 Eliminating Areas of Memory Captured by Core Dumps*, p.366.

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:

rtpName[_*suffix*_]*index*[**.vxcore**]

For more information about file names and locations, see *14.5 Core Dump File Names*, p.367 and *14.6 Core Dump File Locations*, p.367.

File Compression

The RTP core dump facility provides options for file compression (zlib, RLE, or none). For more information, see *14.3 File Compression*, p.366.

Limitations of Core Dumps for VxWorks SMP

The core dump facility has a few limitations with regard to register information and stack traces when used with VxWorks SMP. Note the following behavior with regard to CPUs other than the one that initiated the core dump:

- The contents of the CPU registers and stack trace may not reflect the exact state of a CPU when the core dump was generated. An interprocessor interrupt is used to retrieve information about each CPU, and if interrupts are locked on any of those CPUs when the interrupt occurs, then the contents of the registers and the stack trace may be different when the interrupts are unlocked.
- The contents of a CPU's registers and the stack trace may be missing. If interrupts are locked on a CPU, and then not released while the core dump is being generated, the contents of the registers and the stack trace cannot be retrieved.
- Coprocessor registers are not available.

14.2 VxWorks Configuration for 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
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 *14.4 Eliminating Areas of Memory Captured by Core Dumps*, p.366. For more information on customizing core file names, see *14.5 Core Dump File Names*, p.367)

INCLUDE_CORE_DUMP_RTP_FS
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 (where **remote** is the name of the device; usually a host name). The path must be accessible from the target. By default this parameter is not defined, in which case the core dump is generated in the same location as the RTP binary.

INCLUDE_CORE_DUMP_RTP_FS
Provides functions for getting and setting the storage directory at runtime (for more information, see *14.6 Core Dump File Locations*, p.367), as well as for producing an information file for each RTP core dump file (for more information, see *14.8 Information about RTP Core Dumps*, p.368).

Core Dump Compression

INCLUDE_CORE_DUMP_RTP_COMPRESS_RLE
Provides core dump compression support (based on run-length encoding) that compresses the core dump image. This is the default for RTPs.

INCLUDE_CORE_DUMP_RTP_COMPRESS_ZLIB
Provides a zlib-based alternative to the default, RLE-based, compression method.

CORE_DUMP_RTP_ZLIB_COMPRESSION_LEVEL
Set this parameter to specify 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 *14.3 File Compression*, p.366.

14.3 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-length 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.

14.4 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.

Memory filters are provided by the basic **INCLUDE_CORE_DUMP_RTP** component.

Filters can be installed and removed by the RTP application itself using the **coreDumpMemFilterAdd( )** and **coreDumpMemFilterDelete( )** functions.

Filters are deleted automatically when a RTP application exits.

For more information, see the **coreDumpLib** API reference entry.

14.5 Core Dump File Names

Core dump files are named automatically.

The syntax is as follows:

rtpName[_*suffix*_]*index*[**.vxcore**]

where:

- *rtpName* is the name of the RTP binary without the **.vxe** extension.
- 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 **.vxcore** extension identifies the file as a core dump.

In order to preserve core dump files between sessions, you can use the **coreDumpRtpNameSuffixSet( )** function 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( )**.

14.6 Core Dump File Locations

The location to which core dump files are written is defined statically at system configuration time.

You can, however, use the **coreDumpRtpFsPathGet( )** and **coreDumpRtpFsPathSet( )** functions 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 configuration, see *14.2 VxWorks Configuration for RTP Core Dump Support, p.365*.

14.7 Generating RTP Core Dump Generation

RTP core dumps can be generated automatically or intentionally.

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**.

An RTP application can generate an RTP core dump of itself when, for example, detecting a fatal error that should be debugged.

You can also generate a core dump (from a kernel task or another RTP) by calling **rtpKill( )**. For example:

```
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 *8.1 About Signals*, p.152. For information about the error detection and reporting facilities, see *13. Error Detection and Reporting*. For information about **rtpKill( )**, see the VxWorks API reference entry.

14.8 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/reader_device128_3.vxcore
Size:                    1186905
Time:                    WED MAR 05 11:15:17 2015 (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.

14.9 RTP Core Dump Files Retrieval

How core dump files are retrieved depends on whether you have written them to a local or remote file system.

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 *14.2 VxWorks Configuration for RTP Core Dump Support, p.365*.

14.10 RTP Core Dump Analysis

You can use the Workbench debugger for or **wrshell** for analyzing RTP core dumps.

To successfully debug an RTP core dump, you need the following:

- The generated RTP core dump.
- 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.

15 Safety Profile Facilities for RTP Application Control

15.1 About VxWorks Safety Profile Facilities

The VxWorks 7 Safety Profile provides additional control over individual RTP applications' access system resources.

Access can be restricted for the following resources:

- Public objects, directories, and files.
- Heap memory and shared memory (virtual).
- System calls.
- CPU time.

The VxWorks code for these facilities provided with the VSB **CORE_SAFETY** option. Each facility has its own VIP component.

15.2 RTP Application Specification for Safety Profile Systems

For all Safety Profile facilities, an RTP instance table must be used to specify the RTP applications that will execute in the system. Any attempt to spawn RTPs that have not been defined in the table will fail.

RTP Instance Configuration Table

An RTP instance configuration table consists of type **_Vx_rtp_cfg_t** structures, each of which identifies an RTP application. The structure is defined as follows:

```
typedef struct
    {
    char *              rtpExeName;
    unsigned int        instNum;
    } _Vx_rtp_cfg_t;
```

The first element of each structure specifies the path and executable name. The path must be the same as the one that will be used with the **rtpSpawn( )** call to start the application.

The second element identifies the instance of the application. If only one instance of the application will be spawned, use **0** (zero). If more than one instance will be spawned, increment the count by one for each additional instance (beginning with zero).

In order to associate access control specifications with RTP applications, each of the access control facilities uses tables in which the entries must correspond to the entries in your **_Vx_rtp_cfg_t** application specification table, based on their indices.

To prevent corruption or malicious alteration of access control data structures, Wind River recommends that they be placed in the **.text** or **.rodata** section (and the section made read-only by including the **INCLUDE_PROTECT_TEXT** component), or located in the ROM area.

Example RTP Application Specification

This table identifies three RTP applications, one of which will be instantiated twice (**rtpB.vxe**):

```
LOCAL _Vx_rtp_cfg_t rtpCfg[] =
    {
        {"/romfs/rtpA.vxe", 0},       /* RTP instance index number 0 */
        {"/romfs/rtpB.vxe", 0},       /* RTP instance index number 1 */
        {"/romfs/rtpB.vxe", 1},       /* RTP instance index number 2 */
        {"/romfs/rtpC.vxe", 0}        /* RTP instance index number 3 */
    };
```

15.3 Access Control for Public Objects, Directories, and Files

Access control for public objects, directories, and files provides a mechanism for restricting RTP applications' access to specific public objects (or none), as well as to specific sets of directories and files (or none).

If this facility is included in the system (with **INCLUDE_OBJ_ACCESS_CTRL**), it must be configured to allow RTP applications access to public objects, directories, or files—or to none. That is, access must be defined specifically for each RTP application that will be spawned. If not, the spawn operation will fail.

NOTE: Access control for public objects, directories, and files is provided by the Safety Profile.

15.4 Data Structures for Controlling Access to Public Objects, Directories, and Files

Access control for public object, directories, and files is configured using two tables: one that defines access permissions for public kernel objects, directories, and files; and another that associates access permissions with RTP applications.

To prevent corruption or malicious alteration of access control data structures, Wind River recommends that they be placed in the **.text** or **.rodata** section (and the section made read-only by including the **INCLUDE_PROTECT_TEXT** component), or located in the ROM area.

Table for Public Object, Directory, and File Access Permissions

The table for access permissions consists of **_Vx_obj_access_cfg_t** structure elements, each of which defines access permission for a specific type of public object (or all types), for directories, or files.

```
typedef struct vxObjAccessCfg
    {
    const char * pName;         /* name of the entry */
    _Vx_obj_access_type type;   /* type of the entry */
    _Vx_obj_access_flags flags; /* Optional flags to restrict IO access in
```

```
                                 * O_RDONLY, O_WRONLY or O_RDWR.
                                 * This field is used only for
                                 * VX_ACCESS_CFG_TYPE_IO. Must be set 0 for
                                 * other types.
                                 */
    } _Vx_obj_access_cfg_t;
```

The following macros are used for the *type* element to identify the type of public object (or all public objects), or file, or directory.

Type Macro	Object, Directory, or File
VX_ACCESS_CFG_TYPE_SEMAPHORE	Public semaphore.
VX_ACCESS_CFG_TYPE_SEM_PX	Public POSIX semaphore.
VX_ACCESS_CFG_TYPE_MSG_Q	Public message queue.
VX_ACCESS_CFG_TYPE_MQ_PX	Public POSIX message queue.
VX_ACCESS_CFG_TYPE_TASK	Public task.
VX_ACCESS_CFG_TYPE_TIMER	Public timer.
VX_ACCESS_CFG_TYPE_SD	Public shared data.
VX_ACCESS_CFG_TYPE_CONDVAR	Public conditional variable.
VX_ACCESS_CFG_TYPE_OBJECT_ALL	Any public object—but not directories or files. With this *type* macro, *pName* must point to an * (asterisk) string.
VX_ACCESS_CFG_TYPE_IO	Directory or file.

For public object types (and not directories or files):

- If *pName* points to an * (asterisk) string, access is permitted to all of the public objects specified by *type*.
- *pName* must point to an * (asterisk) string if *type* is set to **VX_ACCESS_CFG_TYPE_OBJECT_ALL**.
- Otherwise *pName* must point to a string with the name of valid public object, the name of which must begin with a **/** (forward-slash).
- The *flags* element must be set to **0** (zero).

For directories and files (**VX_ACCESS_CFG_TYPE_IO** only):

- The string pointed to by *pName* must begin with a **/** (forward-slash) directory delimiter.
- If the *pName* string ends with **/*** (forward-slash and asterisk), access is allowed to contents of the directory—but not to the directory itself, nor its predecessors, nor recursively to the contents of subordinate directories. For example, **/romfs/foo/*** would allow access to the contents of **/romfs/foo**, but not to **/romfs/foo** itself (or **/romfs/foo/**), nor would it allow access to **/romfs/foo/bar/doc.txt**. The access mode for the contents is specified by the *flags* element, except if an individual element is specifically assigned different permissions.

- The *flags* element must be set with one of the access modes: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**.

If these rules are not followed, the RTP application associated with a defective entry will not be spawned.

Public Object, Directory, and File Access Permission Examples

This table allows access to all public objects:

```
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl0 [] =
    {
        {"*", VX_ACCESS_CFG_TYPE_OBJECT_ALL, 0}
    };
```

This table allows access to the **/mySem** public semaphore:

```
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl1 [] =
    {
        {"/mySem", VX_ACCESS_CFG_TYPE_SEMAPHORE, 0}
    };
```

This table allows access to **/ram0** and the subdirectories and files immediately under it, with the access permission for **/ram0** being read-only, and the access permission for all the subdirectories and files immediately under it being read and write:

```
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl2 [] =
    {
        {"/ram0", VX_ACCESS_CFG_TYPE_IO, O_RDONLY},
        {"/ram0/*", VX_ACCESS_CFG_TYPE_IO, O_RDWR}
    };
```

This empty table does not allow access to any public object, directory, or file:

```
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl3 [] =
    {
    /* empty table not to allow to access any public object, directory or file
     */
    };
```

Table for RTP Application Access Control to Public Objects, Directories, and FIles

The table for RTP application access control associates specific RTP applications with a specific set of access permissions. It consists of **_Vx_obj_access_ctrl_t** structure elements, each of which points to an access configuration table and identifies the number of elements in that table.

```
typedef struct vxObjAccessCtrl
    {
    const _Vx_obj_access_cfg_t * pAccessCfgTbl; /* pointer to access
                                                   configuration table */
    unsigned int                 numElement;    /* number of access
                                                   configuration elements */
    } _Vx_obj_access_ctrl_t;
```

The **ACCESS_CTRL_ENTRY** convenience macro can be used in place of the structure. It is defined as follows:

```
#define ACCESS_CTRL_ENTRY(accessCfgTbl) \
    {accessCfgTbl, NELEMENTS(accessCfgTbl)}
```

In order to associate the permissions that you have defined with the appropriate RTP applications, the entries in this table must correspond to the entries in your **_Vx_rtp_cfg_t** application specification table, based on their indices (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372).

Entries must be made even for RTP applications that will not be allowed access to any public object, directory, or file. These entries must point to an empty table (with zero elements).

If an RTP application does not have entry in the access control table associating it with an access permission table, it will not be spawned.

CAUTION: If an RTP application is spawned by a task in another RTP (and not a kernel task), then the spawning RTP application must be given access to the image file for the RTP application that it will spawn. Similarly, an RTP application that calls a shared library must be given access to the shared library file.

RTP Application Access Control Example

The following example defines access for four RTP application instances:

```
LOCAL const _Vx_obj_access_ctrl_t objAccessCtrlTbl [] =
    {
        ACCESS_CTRL_ENTRY(objAccessCfgTbl0),   /* for RTP instance 0 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl1),   /* for RTP instance 1 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl2),   /* for RTP instance 2 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl3)    /* for RTP instance 3 */
    };
```

Each table entry points to an access configuration table. In this example, they point to the tables illustrated in *Public Object, Directory, and File Access Permission Examples*, p.375. Therefore:

- The first element has access all public objects.
- The second has access to the **/mySem** public semaphore.
- The third has access to **/ram0** directory in read-only mode, and access to the sub-directories and files directly under **/ram0** in read and write modes.
- The fourth does not have access to any public object, directory, or file.

The indices for the table entries correspond to the appropriate indices in the application specification table (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372), thereby associating the access permissions with the correct RTP applications.

15.5 Implementing Access Control for Public Objects, Directories, and Files

Implementing access control involves configuring VxWorks with the required components; adding code to the **rtpPartition.c** template file (or a user-defined kernel application) to specify RTP applications, to initialize access control, and to spawn RTPs; and then using **usrAppInit.c** (or the shell) to start the code in **rtpPartition.c**.

Step 1: Create a VSB project, select the **CORE_SAFETY** option, and build the source code.

Step 2: Create a VIP project (based on the VSB project) and add the

INCLUDE_OBJ_ACCESS_CTRL component.

Step 3: If you want to use the **objAccessCtrlShow( )** show function, add **INCLUDE_OBJ_ACCESS_CTRL_SHOW** as well.

Step 4: Save your VIP project.

This adds **rtpPartition.c** to the project. The file includes complete implementation examples.

Step 5: Open **rtpPartition.c**.

Step 6: Add the following include statements for the required header files:

```
#include <vxWorks.h>
#include <rtpLib.h>
#include <rtpPartLib.h>
#include <objAccessCtrlLib.h>
#include <fcntl.h>
```

Step 7: Add an RTP instance configuration table of type **_Vx_rtp_cfg_t** structures to identify the RTP applications (including any multiple instances) that will be executed.

For detailed information, see *15.2 RTP Application Specification for Safety Profile Systems*, p.372.

Step 8: Add an access permissions table to define which public objects, directories, and files your RTP applications should be able to use.

For detailed information, see *Table for Public Object, Directory, and File Access Permissions*, p.373.

Step 9: Add an access control table to associate specific RTP applications with a specific set of access permissions.

For detailed information, see *Table for RTP Application Access Control to Public Objects, Directories, and FIles*, p.375.

Step 10: Create a start-up function that will make all the calls required to initialize the access facility and to spawn the RTPs.

This function is called **usrRtpPartitionInit ( )** in the example in **rtpPartition.c**.

Step 11: In your **usrRtpPartitionInit ( )** function, first call **rtpPartListInit( )** to create the list of RTP instances that will be run.

Step 12: Next in your **usrRtpPartitionInit ( )** function, call **objAccessCtrlConfigure( )** to set the table.

Step 13: Finally in your **usrRtpPartitionInit ( )** function, call **rtpSpawn( )** for each of the RTP applications.

Step 14: Build VxWorks.

Step 15: After booting VxWorks, if you have configured VxWorks with **INCLUDE_OBJ_ACCESS_CTRL_SHOW**, you can use **objAccessCtrlShow( )** to display a list of public objects, directories, and files to which a given RTP application has access.

For additional information, see the comments and example code in the **rtpPartition.c** file in your VIP project, and the VxWorks API references.

rtpPartition.c Example for Controlling Access to Public Objects, Directories, and Files

This example provides an illustration of fully-functional **rtpPartition.c** file that implements access control for four RTP applications—for public objects, directories and files, as well as for no access at all.

```
/* required header files */

#include <vxWorks.h>
#include <rtpLib.h>
#include <rtpPartLib.h>
#include <objAccessCtrlLib.h>
#include <fcntl.h>


/* RTP application specification */

LOCAL const _Vx_rtp_cfg_t rtpCfg[] =
    {
        {"/romfs/rtpA.vxe", 0},      /* RTP instance index number 0 */
        {"/romfs/rtpB.vxe", 0},      /* RTP instance index number 1 */
        {"/romfs/rtpB.vxe", 1},      /* RTP instance index number 2 */
        {"/romfs/rtpC.vxe", 0}       /* RTP instance index number 3 */
};


/* Access permissions for public objects, dirs, files */

LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl0 [] =
    {
        {"*", VX_ACCESS_CFG_TYPE_OBJECT_ALL, 0}
    };
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl1 [] =
    {
        {"/mySem", VX_ACCESS_CFG_TYPE_SEMAPHORE, 0}
    };
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl2 [] =
    {
        {"/ram0", VX_ACCESS_CFG_TYPE_IO, O_RDONLY},
        {"/ram0/*", VX_ACCESS_CFG_TYPE_IO, O_RDWR}
    };
LOCAL const _Vx_obj_access_cfg_t objAccessCfgTbl3 [] =
    {
    /* empty table not to allow to access any public object, directory or file
*/
    };


/* Access control for RTP applications */

LOCAL const _Vx_obj_access_ctrl_t objAccessCtrlTbl [] =
    {
        ACCESS_CTRL_ENTRY(objAccessCfgTbl0),  /* for RTP instance 0 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl1),  /* for RTP instance 1 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl2),  /* for RTP instance 2 */
        ACCESS_CTRL_ENTRY(objAccessCfgTbl3)   /* for RTP instance 3 */
    };


/* Startup function for init functions and spawning RTP apps */

STATUS usrRtpPartitionInit (void)
    {
    int ix;


    if (rtpPartListInit (&rtpCfg[0], NELEMENTS(rtpCfg)) != OK)
        {
        return ERROR;
        }
    if (objAccessCtrlConfigure (&objAccessCtrlTbl[0],
                NELEMENTS(objAccessCtrlTbl)) != OK)
        {
        return ERROR;
        }

    for (ix = 0; ix < NELEMENTS (rtpCfg); ix++)
        rtpSpawn (rtpCfg[ix]. rtpExeName, NULL, NULL, 100, 0x1000, 0, 0);


    return OK;
    }
```

15.6 Access Control for Memory Resources

Memory access control provides a mechanism for limiting the amount memory that RTP applications can use. The memory that is controlled in this way comes from the kernel common heap, the global RAM pool, and the shared memory region virtual memory pool.

Allocations that would result in the RTP exceeding the limit will fail and return an “out of memory” error that is applicable to the allocation type.

If this facility is included in the system (with **INCLUDE_RESOURCE_ALLOC_CTRL**), it must be configured to allow RTP applications access to memory. That is, access to memory must be defined specifically for each RTP application that will be spawned. If not, the spawn operation will fail.

Memory Allocation and Use Calculation

The memory resources that are limited and tracked come from the kernel common heap, global RAM pool, and allocations from the shared memory region virtual memory pool. Note that memory for kernel objects (semaphores, tasks, and so on) that are created by an RTP task are allocated from the heap.

The access control facility provides for separate configuration of heap memory limits and shared memory limits, for each RTP application that will be spawned.

Allocations are tracked for RTP tasks. The only exception is when a kernel task executes **rtpSpawn( )**. The memory allocated by **rtpSpawn( )** is counted toward the RTP that has been created, regardless of whether the parent is a kernel task or an RTP task.

Shared mappings (non-private **mmap** and non-private shared data) are counted against each RTP that maps them, both as physical memory (from the global RAM pool) and virtual memory (from the shared memory virtual page pool). These memory allocations are subtracted from each RTP’s use when they are un-mapped.

When an RTP is deleted it’s memory is returned, and the tracking of it’s memory resources ceases.

NOTE: Access control for memory resources is provided by the Safety Profile.

15.7 Data Structures for Controlling Access to Memory

Access control for memory resource is configured with a table that defines the limits for heap memory and for shared memory use, for each RTP application that will be spawned.

To prevent corruption or malicious alteration of access control data structures, Wind River recommends that they be placed in the **.text** or **.rodata** section (and the section made read-only by including the **INCLUDE_PROTECT_TEXT** component), or located in the ROM area.

Table for Memory Access Permissions

The table for access permissions consists of **_Vx_res_alloc_cfg_t** structure elements, each of which defines limits for memory and for shared memory.

```
typedef struct
    {
    size_t memLimit;
    size_t shmLimit;
    } _Vx_res_alloc_cfg_t;
```

The first member of the structure is used to define the heap memory allocation limit, and the second member is used to define the shared memory allocation limit.

In order to associate the permissions that you have defined with the appropriate RTP applications, the entries in this table must correspond to the entries in your **_Vx_rtp_cfg_t** application specification table, based on their indices (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372).

Memory Access Permissions Example

This table defines heap and shared memory limits for four RTPs.

```
    LOCAL _Vx_res_alloc_cfg_t resAllocCfg[] =
    {
    {0x100000, 0x100000},
    {0x100000, 0x100000},
    {0x2000000, 0x2000000},
    {0x2000000, 0x2000000}
    };
```

The indices for the table entries correspond to the appropriate indices in the application specification table (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372), thereby associating the access permissions with the correct RTP applications.

15.7.1 Implementing Access Control for Memory

Implementing access control involves configuring VxWorks with the required components; adding code to the **rtpPartition.c** template file (or a user-defined kernel application) to specify RTP applications, to initialize access control, and to spawn RTPs; and then using **usrAppInit.c** (or the shell) to start the code in **rtpPartition.c**.

Step 1: Create a VSB project, select the **CORE_SAFETY** option, and build the source code.

Step 2: Create a VIP project and add the **INCLUDE_RESOURCE_ALLOC_CTRL** component.

Step 3: If you want to use the **resAllocCtrlShow ( )** show routine, add **INCLUDE_RESOURCE_ALLOCATION_CTRL_SHOW** as well.

Step 4: Save your VIP project.

This adds **rtpPartition.c** to the project. The file includes complete implementation examples.

Step 5: Open **rtpPartition.c**.

Step 6: Add the following include statements for the required header files:

```
#include <vxWorks.h>
#include <rtpLib.h>
#include <rtpPartLib.h>
#include <resAllocCtrlLib.h>
```

Step 7: Add an RTP instance configuration table of type **_Vx_rtp_cfg_t** structures to identify the RTP applications (including any multiple instances) that will be executed.

For detailed information, see *15.2 RTP Application Specification for Safety Profile Systems*, p.372.

Step 8: Add an access permissions table to define the memory use limits for your RTP applications.

For detailed information, see *15.7 Data Structures for Controlling Access to Memory*, p.380.

Step 9: Create a start-up function that will make all the calls required to initialize the access facility and to spawn the RTPs.

This function is called **usrRtpPartitionInit ( )** in the example in **rtpPartition.c**.

Step 10: In your **usrRtpPartitionInit ( )** function, first call **rtpPartListInit( )** to create the list of RTP instances that will be run.

Step 11: Next in your **usrRtpPartitionInit ( )** function, call **resAllocCtrlConfigure( )** to initialize the access control configuration.

Step 12: Build VxWorks.

Step 13: After booting VxWorks, if you have configured VxWorks with **INCLUDE_RESOURCE_ALLOCATION_CTRL_SHOW**, you can use **resAllocCtrlShow( )** to display information about the RTP memory usage and memory limits.

For additional information, see the comments and example code in the **rtpPartition.c** file in your VIP project, and the VxWorks API references.

rtpPartition.c Example for Memory Access Control

This example provides an illustration of an **rtpPartition.c** file that implements memory access control for four RTP applications.

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

```
#include <resAllocCtrlLib.h>

/* RTP application specification */

LOCAL _Vx_rtp_cfg_t rtpCfg[] =
    {
        {"/romfs/rtpA.vxe", 0},      /* RTP instance index number 0 */
        {"/romfs/rtpB.vxe", 0},      /* RTP instance index number 1 */
        {"/romfs/rtpB.vxe", 1},      /* RTP instance index number 2 */
        {"/romfs/rtpC.vxe", 0}       /* RTP instance index number 3 */
    };


/* Memory limit specifications */

LOCAL _Vx_res_alloc_cfg_t resAllocCfg[] =
    {
    {0x100000, 0x100000},
    {0x100000, 0x100000},
    {0x2000000, 0x2000000},
    {0x2000000, 0x2000000}


/* Startup function for init functions and spawning RTP apps */


STATUS usrRtpPartitionInit (void)
    {
    int ix;

    if (rtpPartListInit (&rtpCfg[0], NELEMENTS(rtpCfg)) != OK)
        {
        return ERROR;
        }


    if (resAllocCtrlConfigure (&resAllocCfg[0], NELEMENTS(resAllocCfg)) != OK)
        {
        return ERROR;
        }
    for (ix = 0; ix < NELEMENTS (rtpCfg); ix++)
        rtpSpawn (rtpCfg[ix]. rtpExeName, NULL, NULL, 100, 0x1000, 0, 0);

    return OK;
    }
```

15.8 Access Control for System Calls

System call access control provides a mechanism for defining the set of system calls to which an RTP application has access.

If this facility is included in the system (with **INCLUDE_SYSCALL_ACCESS_CTRL**), it must be configured to allow RTP applications access to system calls. That is, access to system calls must be defined specifically for each RTP application that will be spawned. If not, the spawn operation will fail.

NOTE: Access control for system calls is provided by the Safety Profile.

15.9 Data Structures for Controlling Access to System Calls

Access control for system calls is configured using two tables: one that defines access permissions for system calls; and another that associates access permissions with RTP applications.

To prevent corruption or malicious alteration of access control data structures, Wind River recommends that they be placed in the **.text** or **.rodata** section (and the section made read-only by including the **INCLUDE_PROTECT_TEXT** component), or located in the ROM area.

Table for System Call Access Permissions

The table for access permissions consists of character strings, each of which defines access permissions for one or more system calls.

Each table can define one of the following basic sets of access permissions:

- All system calls.
- Only those system calls required for spawning an RTP application and for its termination.
- System calls that the application requires in addition to those required for spawning an RTP application and for its termination.

The **SYSCALL_ENTRY_ALL_LIST** macro can be used to allow access to all system calls. It is defined as follows:

```
#define SYSCALL_ENTRY_ALL_LIST "*"
```

System Call Access Permission Examples

The following example provides access to all system calls:

```
LOCAL const char * accessCfgTbl0 [] =
    {
        SYSCALL_ENTRY_ALL_LIST
    };
```

In the next example, macros are used to identify the system calls that are required for spawning an RTP application and for its termination; as well as additional system calls that are required for the RTP application, which are identified by name.

Two configuration tables are defined based on the macros. The first provides access to the system calls required for spawning and termination, as well as to **_write( )**. The second provides access to the system calls required for spawning and termination, as well as to **_write( )** and **_read( )**.

NOTE: This example is provided for illustrative purposes only. The list of system calls identified for spawning and terminating RTPs is dependent on the requirements of the RTP applications.

```
/* Identification of system calls required for spawn and termination */

#define SYSCALL_ENTRY_PROLOGUE_LIST     \
    "mmap",                             \
    "sysctl",                           \
    "objInfoGet",                       \
    "_semOpen",                         \
    "taskCtl",                          \
    "rtpInfoGet"
```

```
#define SYSCALL_ENTRY_EPILOGUE_LIST     \
    "_exit",                            \
    "_close"

#define SYSCALL_ENTRY_MIN_LIST                          \
    SYSCALL_ENTRY_PROLOGUE_LIST, SYSCALL_ENTRY_EPILOGUE_LIST

/* Access configuration tables */

LOCAL const char * accessCfgTbl1 [] =
    {
        SYSCALL_ENTRY_MIN_LIST,
        "_write"
    };

LOCAL const char * accessCfgTbl2 [] =
    {
        SYSCALL_ENTRY_MIN_LIST,
        "_read",
        "_write"
    };
```

Table for RTP Application System Call Access Control

The table for RTP application access control associates specific RTP applications with specific sets of access permissions. It consists of **_Vx_sc_access_ctrl_t** structure elements, each of which points to an access configuration table and identifies the number of elements in that table.

```
typedef struct vxScAccessCtrl
    {
    const char **       pAccessCfgTbl;  /* pointer to access
                                         * configuration table
                                         */
    unsigned int        numElement;     /* number of access
                                         * configuration elements
                                         */
    } _Vx_sc_access_ctrl_t;
```

The **ACCESS_CTRL_ENTRY** convenience macro can be used in place of the structure. It is defined as follows:

```
#define ACCESS_CTRL_ENTRY(accessCfgTbl) \
    {accessCfgTbl, NELEMENTS(accessCfgTbl)}
```

In order to associate the permissions that you have defined with the appropriate RTP applications, the entries in this table must correspond to the entries in your **_Vx_rtp_cfg_t** application specification table, based on their indices (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372).

If an RTP application does not have entry in the access configuration table associating it with an access permission table, it will not be spawned.

RTP Application System Call Access Control Example

The following example defines access for four RTP application instances:

```
LOCAL const _Vx_sc_access_ctrl_t scAccessCtrlTbl [] =
    {
        ACCESS_CTRL_ENTRY(accessCfgTbl0),   /* for RTP instance 0 */
        ACCESS_CTRL_ENTRY(accessCfgTbl1),   /* for RTP instance 1 */
        ACCESS_CTRL_ENTRY(accessCfgTbl1),   /* for RTP instance 2 */
        ACCESS_CTRL_ENTRY(accessCfgTbl2)    /* for RTP instance 3 */
    };
```

Each table entry points to an access configuration table. In this example, they point to the tables illustrated in *System Call Access Permission Examples*, p.383. Therefore:

- The first element allows all system calls.
- The second and third allow access to the system calls required for spawning and terminating the RTP application, and for the **_write( )** system call.
- The fourth allow access to the system calls required for spawning and terminating the RTP application, and for the **_write( )** and **_read( )** system calls.

The indices for the table entries correspond to the appropriate indices in the application specification table (see *15.2 RTP Application Specification for Safety Profile Systems*, p.372), thereby associating the access permissions with the correct RTP applications.

15.10 Implementing Access Control for System Calls

Implementing access control involves configuring VxWorks with the required components; adding code to the **rtpPartition.c** template file (or a user-defined kernel application) to specify RTP applications, to initialize access control, and to spawn RTPs; and then using **usrAppInit.c** (or the shell) to start the code in **rtpPartition.c**.

Step 1: Create a VSB project, select the **CORE_SAFETY** option, and build the source code.

Step 2: Create a VIP project (based on the VSB project) and add the **INCLUDE_SYSCALL_ACCESS_CTRL** component.

Step 3: If you want to use the **syscallAccessCtrlShow( )** show function, add **INCLUDE_SYSCALL_ACCESS_CTRL_SHOW** as well.

Step 4: Save your VIP project.

This adds **rtpPartition.c** to the project. The file includes complete implementation examples.

Step 5: Open **rtpPartition.c**.

Step 6: Add the following include statements for the required header files:

```
#include <vxWorks.h>
#include <rtpLib.h>
#include <rtpPartLib.h>
#include <syscallAccessCtrlLib.h>
```

Step 7: Add an RTP instance configuration table of type **_Vx_rtp_cfg_t** structures to identify the RTP applications (including any multiple instances) that will be executed.

For detailed information, see *15.2 RTP Application Specification for Safety Profile Systems*, p.372.

Step 8: Add an access permissions table to define which system calls your RTP applications should be able to use.

For detailed information, see *Table for System Call Access Permissions*, p.383.

Step 9: Add an access control table to associate specific RTP applications with a specific sets of access permissions.

For detailed information, see *Table for RTP Application System Call Access Control*, p.384.

Step 10: Create a start-up function that will make all the calls required to initialize the access facility and to spawn the RTPs.

This function is called **usrRtpPartitionInit ( )** in the example in **rtpPartition.c**.

Step 11: In your **usrRtpPartitionInit ( )** function, first call **rtpPartListInit( )** to create the list of RTP instances that will be run.

Step 12: Next in your **usrRtpPartitionInit ( )** function, call **syscallAccessCtrlConfigure( )** to initialize the access control configuration

Step 13: Finally in your **usrRtpPartitionInit ( )** function:

- If you want to enable access control before the RTPs are spawned, call **syscallAccessCtrlEnable( )** to start system call access control, then call **rtpSpawn( )** for each of the RTP applications.
- If you want to enable access control after the RTPs are spawned, call **rtpSpawn( )** for each of the RTP applications, then call **syscallAccessCtrlEnable( )** to start system call access control.

Step 14: Build VxWorks.

Step 15: After booting VxWorks, if you have configured VxWorks with **INCLUDE_SYSCALL_ACCESS_CTRL_SHOW**, you can use **syscallAccessCtrlShow( )** to display a list of the system calls that are authorized for the RTP.

For additional information, see the comments and example code in the **rtpPartition.c** file in your VIP project, and the VxWorks API references.

rtpPartition.c Example for System Call Access Control

This example provides an illustration of an **rtpPartition.c** file that implements system call access control for four RTP applications. Preprocessor macros are used to identify the system calls used when an RTP is spawned, and those used when an RTP exits; as well as to control whether access control is started before or after the RTPs have been spawned.

NOTE: This example is provided for illustrative purposes only. The list of system calls identified for spawning and terminating RTPs is dependent on the requirements of the RTP applications.

```
#include <vxWorks.h>
#include <rtpLib.h>
#include <rtpPartLib.h>
#include <syscallAccessCtrlLib.h>

/* RTP application specification */

LOCAL _Vx_rtp_cfg_t rtpCfg[] =
    {
        {"/romfs/rtpA.vxe", 0},       /* RTP instance index number 0 */
        {"/romfs/rtpB.vxe", 0},       /* RTP instance index number 1 */
        {"/romfs/rtpB.vxe", 1},       /* RTP instance index number 2 */
        {"/romfs/rtpC.vxe", 0}        /* RTP instance index number 3 */
};
```

```
/* Identification of system calls required for spawn and termination */

#define SYSCALL_ENTRY_PROLOGUE_LIST     \
    "mmap",                             \
    "sysctl",                           \
    "objInfoGet",                       \
    "_semOpen",                         \
    "taskCtl",                          \
    "rtpInfoGet"

#define SYSCALL_ENTRY_EPILOGUE_LIST     \
    "_exit",                            \
    "_close"

#ifdef ALLOW_RTP_SPAWN_SYSCALLS
#define SYSCALL_ENTRY_MIN_LIST  \
        SYSCALL_ENTRY_EPILOGUE_LIST
#else
#define SYSCALL_ENTRY_MIN_LIST \
    SYSCALL_ENTRY_PROLOGUE_LIST, SYSCALL_ENTRY_EPILOGUE_LIST
#endif

/* Access configuration tables */

LOCAL const char * accessCfgTbl0 [] =
    {
        SYSCALL_ENTRY_ALL_LIST
    };

LOCAL const char * accessCfgTbl1 [] =
    {
        SYSCALL_ENTRY_MIN_LIST,
        "_write"
    };

LOCAL const char * accessCfgTbl2 [] =
    {
        SYSCALL_ENTRY_MIN_LIST,
        "_read",
        "_write"
    };


LOCAL const _Vx_sc_access_ctrl_t scAccessCtrlTbl [] =
    {
        ACCESS_CTRL_ENTRY(accessCfgTbl0),  /* for RTP instance 0 */
        ACCESS_CTRL_ENTRY(accessCfgTbl1),  /* for RTP instance 1 */
        ACCESS_CTRL_ENTRY(accessCfgTbl1),  /* for RTP instance 2 */
        ACCESS_CTRL_ENTRY(accessCfgTbl2)   /* for RTP instance 3 */
    };


/* Startup function for init functions and spawning RTP apps */

STATUS usrRtpPartitionInit (void)
    {
    int ix;

    if (syscallAccessCtrlConfigure (&scAccessCtrlTbl[0],
        NELEMENTS(scAccessCtrlTbl)) != OK)
        {
        return ERROR;
        }

#ifdef ALLOW_RTP_SPAWN_SYSCALLS
/* Allow system calls when RTPs spawned; then enable system call access control
*/
    for (ix = 0; ix < NELEMENTS (rtpCfg); ix++)
         rtpSpawn (rtpCfg[ix]. rtpExeName, NULL, NULL, 100, 0x1000, 0, 0);
    if (syscallAccessCtrlEnable () != OK)
        {
        return ERROR;
        }
#else
/* Enable system call access control; then spawn RTPs */
    if (syscallAccessCtrlEnable () != OK)
        {
        return ERROR;
        }

    for (ix = 0; ix < NELEMENTS (rtpCfg); ix++)
         rtpSpawn (rtpCfg[ix]. rtpExeName, NULL, NULL, 100, 0x1000, 0, 0);
#endif

    return OK;
    }
```

15.11 RTP Time Partition Scheduling

In contrast to the default VxWorks scheduler, the time partition scheduler allows you to schedule RTPs themselves, which guarantees that the RTPs have access to CPU time during a specified time frame. Determinism is insured within each time frame. The time partition scheduler can be used with both uniprocessor and SMP VxWorks.

For information about RTPs and default scheduling, see *RTPs and Scheduling*, p.17.

NOTE: The time partition scheduler is provided by the Safety Profile.

Time Partition Schedule

A time partition schedule (the major frame) consists of a sequence of time windows (minor frames) to which RTP applications and the kernel can be assigned. The schedule repeats after the sum of its window times has passed. Multiple schedules can be defined for a system, and switched dynamically at runtime.

The activation of a time partition schedule and execution of RTP applications occurs as follows:

- VxWorks boots with the default scheduler (only kernel tasks can run at this point).
- The list of RTP applications that will be scheduled is initialized.
- One or more time partition schedules are created.
- The RTP applications are spawned (but do not execute yet).
- The system switches to a time partition scheduler.
- The RTP applications commence execution during their scheduled time windows.
- If more than one time partition schedule has been created, a different one can be used as required, and RTP applications are re-scheduled accordingly. Scheduling can also be returned to the default scheduler.

Time Partition Windows

Each time window in a partition schedule has the following properties:

- A foreground group of RTP applications, whose tasks have precedence over the background group applications. The group may have zero or more members.
- A background group of RTP applications, whose tasks are subordinate to foreground group RTP applications. The group may have zero or more members.
- The duration of the time partition window in ticks (which must be greater than zero).
- Allowance for kernel tasks to run in the foreground RTP group, or the background RTP group, or not at all.

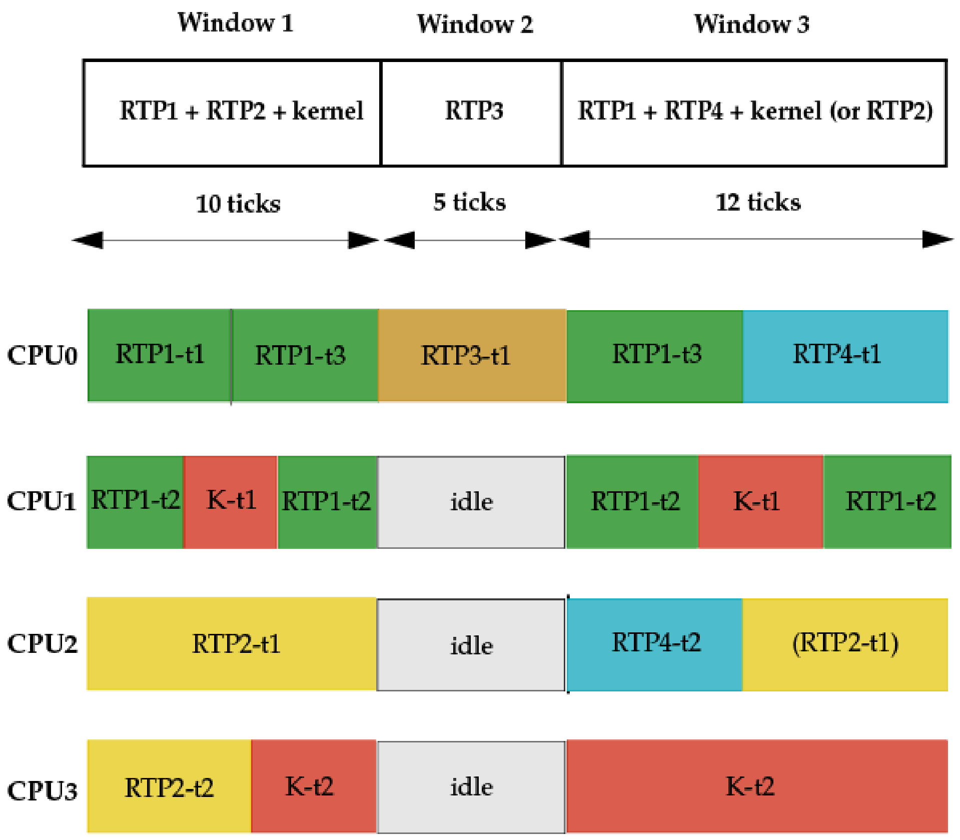
The precedence of foreground group RTP applications means that if any foreground group RTPs' tasks are ready to run, they will run before any background group RTPs' tasks, regardless of tasks priorities. A task in a background RTP can only run if there is no task in a foreground RTP that is ready to run. The background group is useful for activities such as logging, so as not to interfere with more significant applications.

For the details of scheduling behavior, see *15.12 Considerations for Time Partition Scheduling Use*, p.390.

Illustration of a Time Partition Schedule

The following illustration shows a time partition schedule with three time windows, and how tasks might accordingly be scheduled for execution in an SMP multi-core system.

On SMP systems, CPU affinity is set to CPU0 by default with the time partition scheduler. Here it has been explicitly set for the tasks executing on CPU1, CPU2, and CPU3 (see *SMP CPU Affinity and Time Partition Scheduling*, p.392). The parenthesis around RTP2 in Window 3 indicates that it is a background RTP, whose tasks can only run if the tasks in RTP1, RTP4, and the kernel are not ready. Note that RTP2 is also in the foreground group in Window 1.



For information about how this schedule is configured, see the example code in *15.13 Data Structures for Time Partition Schedule Configuration*, p.393.

15.12 Considerations for Time Partition Scheduling Use

Using the time partition scheduler effectively requires consideration of the effects it can have on the behavior of the operating system, as well as a few constraints.

POSIX Thread Scheduler Not Supported

The POSIX thread scheduler for RTPs (**INCLUDE_POSIX_PTHREAD_SCHEDULER**) is not supported with the time partition scheduler because of incompatibilities between the two schedulers.

Task Rotation Not Supported

The task rotation facility (**INCLUDE_TASK_ROTATE**), which provides the **taskRotate( )** function, is not supported with the time partition scheduler because of incompatibilities between these features.

Object Timeouts and Time Partition Windows

An object timeout does not just count the time assigned to the time partition; it is based on the system clock. This means, for example, that if a timeout interval is not accommodated within the time window in which it is initiated, and it expires during another window, the delay is only completed when the initial window is active again. For example, if a timeout of 10 ticks in Window 1 is not run out before Window 2 (with a count of 20 ticks) is active, then the timeout is only completed when Window 1 is again active, which will be more than 20 system clock ticks after initiation.

Time Partition Window Switches on SMP Systems

On SMP systems, time window switches will not occur at the same time across all the enable cores, so that for the duration of the switch, a mix of tasks from the outgoing and incoming time windows will be executing.

Non-Execution of Kernel Tasks

Time partition windows can be configured so that kernel tasks are not allowed to run in them (in either the foreground group or background group). This means, for example, that the interrupt work deferral task, task job deferral task, and log task, are delayed until the next window for which kernel tasks are allowed to run is active. This can lead to lower performance and throughput, lost packages, and so on. You should allow kernel tasks to run in a sufficient number of time partition windows to ensure that the execution of kernel tasks is not delayed for too long.

RTP Executable Accessibility

If an RTP executable image needs to be accessed through network or file system that requires a worker task, but the worker task does not have an execution time window, the RTP will not be spawned successfully.

Resource Dependencies

Resource dependency can lead to idle time partition windows. For example, if a task in the current time window requires access to a shared resource that is protected by a mutex that has been locked in another time window, the current time window may be rendered idle (if it does not include a higher priority task that is ready to run).

Resource dependencies can also affect determinism within a time window. For example, if a low priority task in a background RTP group holds a resource required by a high priority task in a foreground RTP group, and other tasks in the foreground RTP group continue to run, this prevents any background group tasks from running and the resource from being released.

On SMP systems, resource dependencies between tasks running on different CPUs can result in delays in releasing resources because CPU affinity is always in effect. For example, if a task that runs on CPU1 requires access to a shared resource that is protected by a mutex, and the mutex is locked by a task that has CPU affinity with CPU2, the mutex owner cannot run (and release the mutex) as long as there is a higher priority task on CPU2 that is ready to run—even if all other CPUs are in idle loops. See also *SMP CPU Affinity and Time Partition Scheduling*, p.392.

In general, avoid sharing global resources across time partition windows, across RTP groups within a time window, or across CPUs.

Task Scheduling Across RTP Applications

Within a time window, for each foreground and background group, when tasks of the same priority are ready to run at the same time, they are run in the following order:

- First, kernel tasks (if allowed in the time window).
- Then RTP tasks, according to the order in which the RTPs are listed in the configuration of the time partition schedule (see *15.13 Data Structures for Time Partition Schedule Configuration*, p.393).

Each RTP has its own priority queue for ready tasks, and the scheduler iterates over these queues to find the highest priority among the queue heads in the current time window. In the case of multiple tasks of equal priority amongst the queue heads, the first one found is scheduled for execution; that is, they are scheduled sequentially. FIFO order is not guaranteed, and round-robin scheduling of equal priority tasks is therefore less predictable (see *Round-Robin Scheduling*, p.88).

To ensure greater predictability for task execution within a time window, do one of the following:

- Schedule one RTP per time window.
- Use different task priorities in different RTPs.
- Group equal priority tasks in same RTP.

Round-Robin Scheduling and Time Partition Scheduling

Round-robin scheduling can be used with time partition scheduling to share a CPU amongst multiple tasks of a given priority that are ready to run at the same time, within a given time window. With the time partition scheduler, round-robin scheduling does not operate on all tasks of equal priority that are ready to run, but on kernel tasks or RTP tasks separately; according to the order in which tasks of the same priority are executed (see *Task Scheduling Across RTP Applications*, p.392 and *Round-Robin Scheduling*, p.88).

SMP CPU Affinity and Time Partition Scheduling

For VxWorks SMP, adding the time partition schedule (**INCLUDE_TIME_PART_SCHEDULER)** automatically changes the SMP scheduler policy selection to **INCLUDE_SCHED_TIME_PART_POLICY** (replacing **INCLUDE_SMP_SCHED_DEFAULT_POLICY**).

Configuration of a time partition schedule for an SMP system must take into account the time partition scheduler's implementation of CPU affinity, so that scheduling functions on a per-CPU basis.

With the time partition scheduler, all tasks have their CPU affinity set to CPU0 by default. For load balancing, the **taskCpuAffinitySet( )** function must be used assign a tasks CPU affinity to any other CPU. The rules of CPU affinity inheritance are the same as with the default SMP scheduler policy (see *18.10 CPU Affinity*, p.432). Note that with the load balancing you implement, you may for example have a foreground group RTP1 group tasks running on CPU0 and background group RTP2 tasks running on CPU1 at the same time. Also note that task-scheduling across RTP applications as well as round-robin scheduling takes place on a per-CPU basis.

Interrupt and Task Locking

Interrupt locking and task preemption locking delays partition switching. A time partition switch is fully completed once all CPUs in the system start executing tasks that belong to the current time window. If a task from the previous time window delays context switching by locking interrupts with **intCpuLock( )** or by locking task preemption with **taskCpuLock( )**, then completion of the time partition switch is delayed, leading to increased partition switch jitter. Ensure that interrupt locking and task preemption locking are of short duration.

15.13 Data Structures for Time Partition Schedule Configuration

A time partition schedule must be defined with an RTP instance configuration table and a time partition table.

To prevent corruption or malicious alteration of access control data structures, Wind River recommends that they be placed in the **.text** or **.rodata** section (and the section made read-only by including the **INCLUDE_PROTECT_TEXT** component), or located in the ROM area.

Table for RTP Application Specification

An RTP instance configuration table consists of type **_Vx_rtp_cfg_t** structures, each of which identifies an RTP application. The structure is defined as follows:

```
typedef struct
    {
    char *              rtpExeName;
    unsigned int        instNum;
    } _Vx_rtp_cfg_t;
```

The first element of each structure specifies the path and executable name. The path must be the same as the one that will be used with the **rtpSpawn( )** call to start the application.

The second element identifies the instance of the application. If only one instance of the application will be spawned, use **0** (zero). If more than one instance will be spawned, increment the count by one for each additional instance (beginning with zero).

Example Configuration

In the following example, note that both **rtp1.vxe** and **rtp2.vxe** are defined to be instantiated twice.

```
_Vx_rtp_cfg_t rtpCfg[] =
    {
        {"/romfs/rtp1.vxe", 0},         /* index 0 */
        {"/romfs/rtp1.vxe", 1},         /* index 1 */
        {"/romfs/rtp2.vxe", 0},         /* index 2 */
        {"/romfs/rtp2.vxe", 1},         /* index 3 */
        {"/romfs/rtp3.vxe", 0},         /* index 4 */
        {"/romfs/rtp4.vxe", 0}          /* index 5 */
    };
```

The table indices are used to identify the RTP instances in the time partition window specification.

Table for Time Partition Window Specification

A time partition table of type **_Vx_ptw_cfg_t** structures defines the properties of the time partition windows.

Each structure defines the properties of a window, with its elements identifying the following:

- The foreground RTPs (total number and indices; use **0** and **NULL** for none).
- The background RTPs (total number and indices; use **0** and **NULL** for none).
- The number of ticks (must be greater than zero).
- Whether or not kernel tasks are allowed to run, which is specified as follows:
  - **0** to prevent execution of kernel tasks.
  - **TIME_PART_ALLOW_KERNEL_TASKS** to allow kernel tasks to run in the foreground group.
  - **TIME_PART_ALLOW_KERNEL_TASKS_BG** to allow kernel tasks to run in the background group.

(For more information about these properties, see *Time Partition Windows*, p.389.)

Your configuration must conform to the following rules:

- The number of RTP instances defined for the foreground and background groups must be sum of the RTP instances registered for the foreground and background group lists. That is, if a time window specification states that there are two RTP foreground instances in the window, it must also list two specific instances.
- The RTP instances cannot be duplicated—within the foreground group, within the background group, or between the foreground and background groups.
- There must be at least one RTP instance in either foreground or background group unless kernel tasks are allowed to run (with either **TIME_PART_ALLOW_KERNEL_TASKS** or **TIME_PART_ALLOW_KERNEL_TASKS_BG**).

If these rules are not followed, the configuration will be invalid. The schedule creation function **timePartScheduleCreate( )** will return **ERROR**, and **errno** will be set to **S_rtpPart_INVALID_TP_CONFIG**.

Example Configuration

The following example defines three time windows.

```
_Vx_ptw_cfg_t ptwCfg[] =
    {
        {
            2, "0 2",     /* two foreground RTPs: indices for RTP1 and RTP2 */
            0, NULL,      /* no background RTPs */
            10,           /* duration: 10 ticks */
            TIME_PART_ALLOW_KERNEL_TASKS /* kernel tasks allowed */
        },
        {
            1, "4",       /* one foreground RTP: index for RTP3 */
            0, NULL,      /* no background RTPs */
            5,            /* duration 5 ticks */
            0             /* kernel tasks not allowed */
        },
        {
            2, "1 5",     /* two foreground RTPs: index for RTP0, second
                             instance,and RTP 4 */
            1, "3",      /* one background RTP: index for RTP2, second instance
                          */
            12,          /* duration 12 ticks */
            TIME_PART_ALLOW_KERNEL_TASKS /* kernel tasks allowed */
        },
    };
```

Together with the code that identifies the RTP applications in *15.2 RTP Application Specification for Safety Profile Systems*, p.372, this window specification corresponds to the illustration provided in *Illustration of a Time Partition Schedule*, p.389.

15.14 Implementing a Time Partition Schedule

Implementing a time partition schedule involves configuring VxWorks with the required components; adding code to the **rtpPartition.c** template file (or a user-defined kernel application) to define the schedule and initialize it, and to start the RTP applications; and using **usrAppInit.c** (or the shell) to run the code in **rtpPartition.c**.

Step 1: Create a VSB project, add the **CORE_SAFETY** layer, and rebuild the project.

Step 2: Create a VIP project (based on the VSB project) and configure it as follows:

- Add the **INCLUDE_TIME_PART_SCHEDULER** component. (For VxWorks SMP, adding **INCLUDE_TIME_PART_SCHEDULER** automatically changes the SMP scheduler policy selection to **INCLUDE_SCHED_TIME_PART_POLICY**.)
- For show function support, add **INCLUDE_TIME_PART_SHOW**. See *Time Partition Information Functions*, p.397.
- For time partition switch hook support, add **INCLUDE_TIME_PART_SWITCH_HOOKS**. See *Time Partition Switch Hook Functions*, p.398.
- For hook function show function support, add **INCLUDE_TIME_PART_SWITCH_HOOKS_SHOW**.
- If you want to be able to start the time partition scheduler automatically at boot time, add **INCLUDE_USER_APPL**.

Step 3: Save your VIP project.

This adds **rtpPartition.c** to the project. The file includes complete implementation examples.

Step 4: Open **rtpPartition.c**.

Step 5: Use an RTP instance configuration table of type **_Vx_rtp_cfg_t** structures to identify the RTP applications (including any multiple instances) that will be executed.

For detailed information, see *Table for RTP Application Specification*, p.393. Note that the table indices are used to identify the RTP instances in the next step.

Step 6: In **rtpPartition.c**, use a time partition table of type **_Vx_ptw_cfg_t** structures to define the properties of each time partition window.

For detailed information, see *Table for Time Partition Window Specification*, p.394.

Step 7: In **rtpPartition.c**, create a function to initialize and start the partition schedule, as well as to spawn the RTP applications.

This function is called **usrRtpPartitionInit ( )** in the example in **rtpPartition.c**.

Step 8: In your **usrRtpPartitionInit ( )** function, first call **rtpPartListInit( )** to create the list of RTP instances that will be used by the time partition scheduler.

The the first argument is the address of the RTP instance configuration table (**_Vx_rtp_cfg_t)** and the second element is the number of elements in the table. For example:

```
rtpPartListInit (&rtpCfg, NELEMENTS(rtpCfg));
```

Step 9: Next in your **usrRtpPartitionInit ( )** function, call **timePartScheduleCreate( )** to create the schedule.

The first argument is the address of the time partition table (**_Vx_ptw_cfg_t**) and the second is the number of windows (elements in the table). For example:

```
schedId = timePartScheduleCreate (&ptwCfg[0], 4);
```

Step 10: Then in your **usrRtpPartitionInit ( )** function, call **rtpSpawn( )** to start each of the RTP applications.

At boot time the default scheduler only allows kernel tasks to run. While this scheduler is in effect, RTPs can be spawned by kernel tasks, but they will not be executed until a schedule is initiated by a **timePartScheduleSet( )** call.

Step 11: Finally in your **usrRtpPartitionInit ( )** function, call **timePartScheduleSet( )** to initiate use of the schedule.

The first argument is the schedule ID returned by **timePartScheduleCreate( )**, and the second identifies when the schedule should take effect. For example:

```
timePartScheduleSet (schedId, TIME_PART_SCHED_CHANGE_NOW);
```

Note that **timePartScheduleSet( )** can also be used to switch back to the default scheduler (without time partitions).

Step 12: In your VIP project, use **usrAppInit.c** to call your **usrRtpPartitionInit ( )** function automatically at boot time (this requires the **INCLUDE_USER_APPL** component). Alternatively, it may be useful during development to simply call **usrRtpPartitionInit ( )** from the shell.

For information about how to start your kernel application automatically at boot time, see *1.8 Using userAppInit( ) to Run Applications Automatically*, p.11.

Step 13: Rebuild VxWorks.

15.14.1 Using Multiple Time Partition Schedules

You can create multiple time partition schedules, and switch between them at runtime. Wind River recommends that you do so sparingly.

To use multiple schedules:

Step 1: Create additional schedules in the same manner that you would create a single one.

Step 2: Use the **timePartScheduleSet( )** function to initiate the first schedule, and then to switch to another. The first argument is the schedule ID returned by **timePartScheduleCreate( )**, and the second specifies when the schedule should take effect—immediately, at the end of the current window, at the end of the schedule, or after a specified number of ticks.

Note that **timePartScheduleSet( )** can also be used to switch back to the default scheduler (without time partitions).

15.15 Time Partition Scheduling APIs

The time partition scheduler facility provides APIs for creating and managing a schedule, for getting information about the schedule, and for adding and deleting user-defined functions.

Time Partition Management and Information Functions

Function	Description
rtpPartListInit( )	Create the RTP instance list used for time partitioned scheduling.
timePartScheduleCreate( )	Create a new time partitioning schedule.
timePartScheduleSet( )	Set current time partitioning schedule.
timePartCurrentInfoGet( )	Get the remaining and elapsed times in ticks.

For more information, see the **rtpPartLib** and **timePartLib** kernel API references.

Time Partition Information Functions

Function	Description
timePartShow( )	Display information about a specific time window.
timePartShowAll( )	Display information about all the time partition windows.
rtpPartInstShow( )	Display time partition information for all of RTPs that are initialized with **rtpPartListInit( )**.

For more information, see them **timePartShow** and **rtpPartShow** kernel API reference.

Time Partition Switch Hook Functions

Function	Description
timePartSwitchHookAdd( )	Add a function to be called every time a partition switch occurs.
timePartSwitchHookDelete( )	Delete a previously added time partition switch function.
timePartSwitchHookShow( )	Show the list of time partition switch functions.

For more information, see the **timePartSwitchHookLib** and **timePartHookShow** kernel API references.

16
User Authentication and Management

16.1 About User Authentication and Management

The VxWorks user authentication and management facility provides for securing access to a VxWorks system.

The user management facility can be used for all services in VxWorks for which you need to authenticate users. It is particularly intended for services offering login capabilities such as the shell (console, telnet, rlogin, and SSH login), FTP, and SFTP.

Once a user is logged in, the service that user is interacting with (for example, the shell or the FPT server), is automatically owned by that user. That is, the service task's user ID and group ID are set to those of the user (and cannot be changed unless the user is root). With the user privileges feature, users can also be restricted to specific operations of these services. For information about user and group IDs, see *RTP Ownership and Inheritance*, p. 14 and *6.3 Task Ownership and Inheritance*, p. 81.

16.2 VxWorks Configuration

Configuration for user authentication and management involves selection of both VSB and VIP features.

VSB Configuration

Select **USER_MANAGEMENT**.

User and group identification is included in VxWorks by default (with **USER_IDENTIFICATION_INHERITANCE**).

VIP Configuration

For the user database facility add the **INCLUDE_USER_DATABASE** component to a VIP project based on your VSB project. For information about setting the associated parameters, see *16.3 User Database*, p.400. In addition, add file system support for the user database file.

For the shell and login facility, add the **INCLUDE_SHELL** and the **INCLUDE_SHELL_SECURITY** components. For a deployed system you should add the **INCLUDE_STANDALONE_SYM_TBL** component (which links the symbol table to the kernel).

The **INCLUDE_UDB_CUSTOM_TAMPERING_POLICY** component can be used if you want to provide a custom user database tampering handler. See *16.6 Custom Tampering Policy*, p.405.

The **INCLUDE_UDB_CUSTOM_MAC_SUPPORT** component can be used if you want to use a custom wrapper MAC generation function. See *16.7 Using a Custom MAC Generation Function*, p.406.

16.3 User Database

The user database is a text file that stores user account records.

If it does not already exist, database is created automatically when the first user logs on and is prompted to create their password. Once the database is created, it is maintained across reboot and on/off cycles. A maximum of 65,534 users are permitted. The user database facility is provided with the **INCLUDE_USER_DATABASE** component.

The user database manager ensures the run-time safety and security of the information kept in the database. This is done by generating Hashed Message Authentication Code (HMAC) for each of the records in the user database. HMACs are computed based on the content of the user record, so that any tampering with a user record can be detected. By default, standard Base64 encoding and the SHA-256 hash function are used to compute the user record's HMAC. For information about using a custom facility, see *16.7 Using a Custom MAC Generation Function*, p.406.

Location Configuration

The database should be stored on a non-volatile storage device. This storage device should have a configurable file system, making it possible for the manager of the user database to access it through regular file I/O operations.

The location at which the user database is created (automatically) is determined by the **UDB_STORAGE_PATH** configuration parameter, which is associated with the **INCLUDE_USER_DATABASE** component. This parameter's default value is NULL, so it must be set to a valid path, and the file name.

For example, assuming use of the HRFS file system on a USB flash drive that is automatically mounted (**/bd0**), and that the user database file should be named **vxUserDB.txt**, then the following would be assigned to **UDB_STORAGE_PATH**:

"/bd0/vxUserDB.txt"

⚠ **CAUTION:** If you use a mass storage device to store the user database text file, take into consideration that the storage device may not be ready when the user database manager is initialized. If the mass storage device is not up by the time the user database manager is initialized, the system displays an error message such as **Cannot find user database!** and prevents you from adding other users or changing passwords. Once the system has fully booted and the storage device is ready, you can use the database.

Hash Key Configuration

You must specify the secret hash key for computing the HMAC with the **UDB_HASH_KEY** and **UDB_HASH_KEY_LEN** configuration parameters. The default value for the parameter **UDB_HASH_KEY_LEN** is 256. The hash key should have random bytes encoded in *\xnn* format, where *nn* can vary from 00 to ff. The longer the hash key, the higher the quality of cryptographic hashing.

For example, the following could be assigned to **UDB_HASH_KEY** (without line breaks):

```
"\x48\x61\x6d\x6c\x65\x74\x2e\x20\x54\x6f\x20\x62\x65\x2c\x20\x6f\x72\x20\x6e\x6
f\x74\x20\x74\x6f\x20\x62\x65\x2d\x20\x74\x68\x61\x74\x20\x69\x73\x20\x74\x68\x6
5\x20\x71\x75\x65\x73\x74\x69\x6f\x6e\x3a\x0a\x57\x68\x65\x74\x68\x65\x72\x20\x2
7\x74\x69\x73\x20\x6e\x6f\x62\x6c\x65\x72\x20\x69\x6e\x20\x74\x68\x65\x20\x6d\x6
9\x6e\x64\x20\x74\x6f\x20\x73\x75\x66\x66\x65\x72\x0a\x54\x68\x65\x20\x73\x6c\x6
9\x6e\x67\x73\x20\x61\x6e\x64\x20\x61\x72\x72\x6f\x77\x73\x20\x6f\x66\x20\x6f\x7
5\x74\x72\x61\x67\x65\x6f\x75\x73\x20\x66\x6f\x72\x74\x75\x6e\x65\x0a\x4f\x72\x2
0\x74\x6f\x20\x74\x61\x6b\x65\x20\x61\x72\x6d\x73\x20\x61\x67\x61\x69\x6e\x73\x7
4\x20\x61\x20\x73\x65\x61\x20\x6f\x66\x20\x74\x72\x6f\x75\x62\x6c\x65\x73\x2c\x0
a\x41\x6e\x64\x20\x62\x79\x20\x6f\x70\x70\x6f\x73\x69\x6e\x67\x20\x65\x6e\x64\x2
0\x74\x68\x65\x6d\x2e\x20\x54\x6f\x20\x64\x69\x65\x2d\x20\x74\x6f\x20\x73\x6c\x6
5\x65\x70\x2d\x0a\x4e\x6f\x20\x6d\x6f\x72\x65\x3b\x20\x61\x6e\x64\x20\x62\x79\x2
0\x61\x20\x73\x6c\x65\x65\x70\x20\x74\x6f\x20\x73\x61\x79\x20\x77"
```

⚠ **CAUTION:** For security reasons, do not use the hash key provided in this example. It is for illustrative purposes only. Using this hash key may compromise the security of your system.

⚠ **CAUTION:** If the **INCLUDE_SECURITY** component is included in a VIP, but the **UDB_STORAGE_PATH** or **UDB_HASH_KEY** parameters are left set to null, the user database manager will not be initialized properly and the user database will not be created.

User Database Creation

The user database does not initially exist until the first user logs in, at which point it is created automatically by the user management facility at the location specified with **UDB_STORAGE_PATH** configuration parameter.

When the first user logs in, that user is prompted to provide a password—the system does not challenge the user. Any users who subsequently attempt to log on are challenged, because it is assumed that the first user is the "administrator," whose task is to set up the other users' accounts, including their initial passwords.

User Database Management

The following functions can be used for database management.

Function	Description
udbInfoGet( )	Get information about the user database.
udbReset( )	Reset the user database.
udbInfoShow( )	Show information about the user database
udbRepair( )	Attempt to repair the user database.

For more formation about these and additional functions, see the **udbMgr** API reference.

16.4 User Account Management

When a user account is created, a unique user ID and group ID are associated with it, either automatically or explicitly.

If they are not specified by the account creator, the user management facility automatically assigns a unique user ID (starting with 2) and a default group ID (100). The default group ID is set with the **DEFAULT_PRIMARY_GROUP_ID** configuration parameter.

Initial User

If the shell login facility is included in the system (with **INCLUDE_SHELL_SECURITY**), the first user to log on—or the first to log on when the user database is empty—is considered the *initial user*. The initial user is prompted for their password as part of logging on, and assigned the user ID of 2 (zero is not valid and 1 is reserved for the root user). The initial account is essentially an "administrator" account, as it is used to create all other accounts.

If the login facility is not included, the initial user account is not automatically created, and all users must be registered from the shell with the **useradd** command with the command interpreter, or the **userAdd( )** function with the C interpreter (the function can also be used programmatically).

⚠ **CAUTION:** For security reasons, do not deploy a system without having created an initial user. Otherwise, whoever logs on first will get full access to the system.

Additional Users

All additional user accounts should be created by the initial user. By default, the system automatically sets the user ID associated with each newly registered user. The value of the user ID is an increment of 1 from the previously assigned user ID. The user ID cannot be changed. Use the **useradd** command or the **userAdd( )** function to create a new user account.

When you create any user account, you must provide a password to associate with that account. The user can update this password later. For example:

```
-> userAdd "moimoi","foo_passwd"

-> userPasswordUpdate "moimoi", "foo_passwd", "foo_new_passwd"
```

Root User

A root user (super user) can be created, using the user ID of 1 and the group ID of 1. For example:

```
-> userAdd "root","root_passwd", 1, 1
```

The name **root** is merely a useful convention that has no functional significance.

Creation of a root user is only significant when restrictions are placed on user privileges (see *16.5 Defining User Privileges*, p. 404), in which case the root user has permissions for all commands.

User Account Management Functions and Shell Commands

The following functions and commands can be used to manage user accounts. Use the commands from the shell's command interpreter, and the functions from the C interpreter or programmatically.

Function	Command	Description
userAdd( )	**useradd**	Create a new user account.
userDelete( )	**userdel**	Delete a user account.
userAuthenticate( )	**-**	Authenticate a user name and password.
userPasswordUpdate( )	**passwd**	Change a password.
userAuthBlock( )	**-**	Block user authentication.
userAuthUnblock( )	**-**	Unblock user authentication.
getpwuid( )	-	Search the user database for a user ID.
getpwuid_r( )	-	Search the user database for a user ID (reentrant version).

Function	Command	Description
getpwnam( )	-	Search the user database for a name.
getpwnam_r( )	-	Search the user database for a name (reentrant version).

For more information see the **userIdentLib** and **userMgtShellCmd** references.

16.5 Defining User Privileges

By default, user privileges for the kernel shell (as well as FPT and SFPT) are not limited. They can, however, be limited by defining execution rights for elements of these services on a user or group basis with the user privileges facility.

A privileges manifest file defines user privileges. Among other default restrictions defined in the manifest, the kernel shell's C interpreter is disabled and the command interpreter must be used until the manifest is edited. This provides a basic level of initial security by preventing unlimited access to all kernel functions.

Note that the root user is not subject to any restrictions, regardless of the privileges manifest.

For the security of a deployed system, the privileges manifest file can be stored in ROMFS (a read-only file system; see the *VxWorks 7 File Systems Programmer's Guide*.) During development it can be stored in the host's file system.

NOTE: This feature is provided by the Security Profile.

To define user privileges, perform the following operations:

Step 1: Create a VSB and select **USER_MANAGEMENT** and **USER_PRIVILEGES**.

Step 2: Build the VxWorks source code.

Step 3: Create a VIP based on your VSB.

Step 4: Add the following components:

- **INCLUDE_USER_IDENTIFICATION**
- **INCLUDE_USER_DATABASE**
- **INCLUDE_SHELL**
- **INCLUDE_SHELL_SECURITY**

Step 5: For a deployed system also add **INCLUDE_STANDALONE_SYM_TBL**.
This will link the symbol table with the kernel.

Step 6: Set the **UDB_STORAGE_PATH** parameter to the location of the user database.
See *16.3 User Database*, p. 400.

Step 7: Set the **UDB_HASH_KEY** and **UDB_HASH_KEY_LEN** parameters to the hash key and length of the hash key.

See *16.3 User Database*, p.400.

Step 8: Build VxWorks.

Step 9: Boot VxWorks and register all the users who are expected to log on to the system.

See *16.4 User Account Management*, p.402.

Step 10: Add **INCLUDE_USER_PRIVILEGES** to your VIP.

Step 11: Rebuild VxWorks.

This creates a **privilege_manifest** directory in the VIP, which contains the default privilege manifest file, **prvlgManifest.txt**.

Step 12: Edit the **prvlgManifest.txt** file so that it defines the appropriate privileges for the users. For instructions in this regard, see the comments at the top of the file.

If you wish to retain a copy of the **prvlgManifest.txt** file after deleting the VIP project, make a copy of it outside of the project.

Step 13: If you are going to store **prvlgManifest.txt** in the ROMFS file system (which is recommended for a deployed system):

- Add **INCLUDE_ROMFS** to the VIP.
- Add **prvlgManifest.txt** to **/romfs**.

By default the **PRIVILEGE_MANIFEST_PATH** parameter is set to **/romfs/prvlgManifest.txt**, so it does not have to be changed.

Step 14: If you are not going store **prvlgManifest.txt** in ROMFS:

- Copy it to the appropriate location on the target or host system.
- Set the **PRIVILEGE_MANIFEST_PATH** parameter to the location of the file. (By default it is set to **/romfs/prvlgManifest.txt**.)

Step 15: Rebuild VxWorks.

For additional information, see the **privilegesLib** API reference entry, which also describes the **privilegesCheck( )** function.

16.6 Custom Tampering Policy

A custom tampering policy handler with the user database.

The information in the user database is protected through HMACs for each record in the user database (see *16.3 User Database*, p.400). Because these HMACs are computed based on the content of the user record, any tampering can be detected. If there is a discrepancy, the tampering policy handler is executed. The default handler blocks user authentication and logs tampering detection events to the error detection and reporting log, if this feature is included in the system (see *13. Error Detection and Reporting*).

If you want to implement a custom tampering policy handler, use the **INCLUDE_UDB_CUSTOM_TAMPERING_POLICY** component. For detailed information, see the **udbMgr** API reference.

16.7 Using a Custom MAC Generation Function

You can write your own wrapper MAC generation function instead of using the default.

NOTE: This feature is provided by the Security Profile.

If you choose to write your own wrapper MAC generation function instead of using the default (see *16.3 User Database*, p.400), you must perform the following steps. In all other respects you must configure and build your system as described in *VSB Configuration*, p.400 and *VIP Configuration*, p.400.

Step 1: Write your MAC-generation function using the MAC-computing function and the cryptographic hash function of your choice, as well as a Base64-encoding function.

You can use your own encoding function, or the **udbBase64Encode( )** function provided by Wind River (see the *VxWorks Kernel API Reference)*. For an example of a custom MAC generation function, see the **udbMgr** entry in the *VxWorks Kernel API Reference*. Note that a VSB option can be used in code by adding the **_WRS_CONFIG_** prefix. For example, to manipulate the value set with the **UDB_MAC_SZ** VSB option, you must use the **_WRS_CONFIG_UDB_MAC_SZ** macro.

Step 2: Set the VSB options **UDB_MAC_SZ** and **UDB_BASE64_MAC_SZ** to the size of the raw MAC value that will be generated by the function, and to the size of the buffer required for Base64 encoding.

The user database manager manipulates various buffers and size information based on these values, so they must be set properly. If they are not, there will be execution failures.

Step 3: Rebuild your VSB project.

Step 4: Add **INCLUDE_UDB_CUSTOM_MAC_SUPPORT** to your VIP project. This adds **udbCustomMacCompute.c** to the project and sets the **UDB_CUSTOM_MAC_COMPUTE_RTN** parameter to **udbCustomMacCompute( )**.

Step 5: Add your code to the **udbCustomMacCompute( )** function in **udbCustomMacCompute.c**.

Note that **udbCustomMacCompute.c** is deleted when the VIP to which it was added is deleted, so you may wish keep a copy of it outside of the project.

The user database manager will call the **udbCustomMacCompute( )**function whenever it needs to computes a MAC for an entry in the user database.

Step 6: Rebuild your VIP project.

17 UEFI Boot Loader and Secure Boot

17.1 About the UEFI Boot Loader and Secure Boot

A VxWorks system can be created with secure boot capabilities for Intel targets using the UEFI boot loader and firmware.

NOTE: This feature is provided by the Security Profile.

This first involves creating a VxWorks image that is digitally signed, so that the origin and authenticity of the image can be confirmed and so that the target machine can be prevented from running unauthorized software, as well as encrypted to thwart attempts to disassemble or reverse-engineer the user's code.

WIBU Systems provides the host tools and runtime libraries (which are integrated with the VxWorks installation) used to create the VxWorks image. For information in this regard, see *CodeMeter Embedded in VxWorks—Basic Security*. See also *CodeMeter PKI for VxWorks*.

In order to be able to decrypt and validate the protected VxWorks image, the UEFI loader must use the same symmetric key used to encrypt the image and the public key of the root certificate. (This is the same functionality that VxWorks must have to be able to decrypt and validate protected kernel and RTP applications.)

In order to create a fully trusted secure boot system from the firmware up, the target system’s UEFI secure boot support (if available) must be appropriately configured as well.

For more information about the UEFI boot loader, see the *VxWorks 7 BSP and Driver Guide.*

17.2 Creating a UEFI Boot Loader

To create a UEFI boot loader than can load the signed and encrypted VxWorks image, the symmetric key used to encrypt the image and the public key of the root certificate must be compiled into the UEFI loader image, as they are not stored in the target hardware.

For information about key creation, see *CodeMeter Embedded in VxWorks - Basic Security*.

The **BOOT_SECURE make** macro must also be enabled. This builds the **libcm.a** library (which contains the WIBU basic security modules), as well as the decryption and signature validation functionality in the ELF loader code (in *installDir***/vxworks-7/pkgs/boot/uefi-1.0.1.0/common/libelf/elf.c**). For more information about the **BOOT_SECURE** macro, see *17.3 UEFI Secure Boot*, p. 409.

Step 1: Set environment variables in a shell:

```
% wrenv -p vxworks-7
```

Step 2: Change directory to *installDir***/vxworks-7/pkgs/boot/uefi-1.0.1.0**.

Step 3: Enable the **BOOT_SECURE** and (optionally) the **BOOT_FORCE_SECURE** features.

This can be done either With the **make ADDED_FLAGS** variable. For example:

```
ADDED_CFLAGS="-DBOOT_SECURE -DBOOT_FORCE_SECURE"
```

Or by editing the **defs.boot** file in *installDir***/vxworks-7/pkgs/boot/uefi-1.0.2.0** to un-comment these features.

Step 4: Specify the path to the VxWorks VSB project that holds the **exengine_keys.h** file, or copy the file to *installDir***/workspace**.

To specify the path to a VSB project that is in the default *installDir***/workspace** directory, use a **VSB=***vsbname* **make** variable assignment. To specify the path to a VSB project that is not in the default directory, use a **VSB_PATH=***/path/to/vsbname* variable assignment.

(For information about key creation and **exengine_keys.h**, see *CodeMeter Embedded in VxWorks - Basic Security*.)

Step 5: Build the boot loader.

For example:

```
% make ADDED_CFLAGS="-DBOOT_SECURE -DBOOT_FORCE_SECURE" VSB=secureBootVsb
```

The build produces the following loader images:

*installDir***/vxworks-7/workspace/uefi_ia32/BOOTIA32.EFI**

*installDir***/vxworks-7/workspace/uefi_x86_64/BOOTX64.EFI**

You may use either or both in your boot media, as your system requires.

Note that when you create a USB boot drive, you must use the **vxWorks** ELF image and not **vxWorks_romCompress.bin**. (The loader will reject flat binary images and binary images with multi-boot headers, as these are not handled by the signing tools.) Once the disk is formatted, the UEFI loader and VxWorks image may be installed as in this example:

```
E:\> mkdir EFI
```

```
E:\> mkdir EFI\BOOT
E:\> copy C:\path\to\BOOTIA32.EFI EFI\BOOT
E:\> copy C:\path\to\BOOTX64.EFI EFI\BOOT
E:\> copy C:\path\to\protected\vxWorks EFI\BOOT\bootapp.sys
```

17.3 UEFI Secure Boot

If the target includes UEFI secure boot support, then you must also enroll the Platform Key (PK), Key Exchange Key (KEK), and whitelist/blacklist keys with the firmware and digitally sign the **BOOTIA32.EFI** and **BOOTX64.EFI** images so that the firmware will accept them.

The UEFI digital signature scheme is described in the UEFI 2.3.1 and later specification documents. It is, however, different from the WIBU digital signature scheme and requires different tools. Currently Wind River does not provide tools for this purpose, and you must use the Microsoft Windows SDK for UEFI secure boot signing.

If the target has UEFI firmware with secure boot support, and the firmware is operating in secure mode, only signed UEFI applications are permitted to run. The **SecureBoot** environment variable is set to 1 to indicate that the firmware is in secure mode, or to 0 to indicate that secure boot support is disabled. If an application is allowed to execute, it can test this variable to check the security state of the system.

Not all UEFI-enabled boards that are currently available include UEFI firmware builds with secure boot support present. If the UEFI firmware on the system does not include secure boot support, the **SecureBoot** environment variable is not present.

By default, if the VxWorks UEFI loader is built with the **BOOT_SECURE** macro enabled, it will check for the **SecureBoot** environment variable. If the variable is not found or set to 0, the loader will load both protected and unprotected VxWorks images. If the **SecureBoot** variable is found and it's set to 1 (indicating that the firmware is in secure mode and that the loader has been digitally signed), then the loader will only load protected VxWorks ELF images. Unprotected ELF images will not be allowed to run. The loader will also reject flat binary images and binary images with multi-boot headers, as these are not handled by the signing tools.

The **BOOT_FORCE_SECURE** macro is provided to force the loader to only accept protected VxWorks images regardless of whether or not the target system supports UEFI secure boot or if it's disabled.

Note that if the UEFI firmware does not support secure boot, then it's possible for someone to tamper with the UEFI loader code so as to disable the digital signature checks, and to run the patched UEFI loader on the target system. This would allow an attacker to tamper with the VxWorks image and have the loader run it even though the tampering would invalidate the digital signature. This caveat only applies if the image is signed but not encrypted. An attacker would have to decrypt the image first in order to patch the instructions in it.

Also note that since the symmetric encryption key is compiled into the loader, it is possible for an attacker to eventually recover it from the loader binary.

18
VxWorks SMP

18.1 About VxWorks SMP

VxWorks SMP is a configuration of VxWorks designed for symmetric multiprocessing (SMP). It provides the same distinguishing RTOS characteristics of performance and determinism as uniprocessor (UP) VxWorks. The differences between SMP and UP VxWorks are limited, and strictly related to support for multiprocessing.

Multiprocessing systems include two or more processing units in a single system. Symmetric multiprocessing (SMP) is a variant of multiprocessing technology in which one instance of an operating system uses more than one processing unit, and in which memory is shared. An asymmetric multiprocessing (AMP) system, on the other hand, is one in which more than one instance of an operating system—whether SMP or uniprocessor (UP)—is running at the same time.

Terminology

The terms *CPU* and *processor* are often used interchangeably in computer documentation. However, it is useful to distinguish between the two for hardware that supports SMP. In this guide, particularly in the context of VxWorks SMP, the terms are used as follows:

CPU
: A single processing entity capable of executing program instructions and processing data (also referred to as a *core*, as in *multicore*). May refer to a single simultaneous multi-threading (SMT) processing unit on superscalar processors, as well as a core.

  A CPU can be identified by its CPU ID, physical CPU index, and logical CPU index. A *CPU ID* is defined by the systems firmware or hardware. The *physical CPU* indices begin with zero for the system's bootstrap CPU and are incremented by one for each additional CPU in the system. The *logical CPU indices* are specific to the operating system instance. For example, a UP instance always has a logical CPU index of zero; an SMP instance configured for four CPUs assigns logical indexes to those CPUs from 0-3, regardless of the number or how many CPUs are in the physical system.

processor
: A silicon unit that contains one or more CPUs.

multiprocessor
: A single hardware system with two or more processors.

uniprocessor
: A silicon unit that contains a single CPU.

For example, a dual-core MPC8641D would be described as a processor with two CPUs. A quad-core Broadcom 1480 would be described as a processor with four CPUs.

Uniprocessor code may not always execute properly on an SMP system, and code that has been adapted to execute properly on an SMP system may still not make optimal use of symmetric multiprocessing. The following terms are therefore used to clarify the state of code in relation to SMP:

SMP-ready
: Runs correctly on an SMP operating system, although it may not make use of more than one CPU (that is, does not take full advantage of concurrent execution for better performance).

SMP-optimized
: Runs correctly on an SMP operating system, uses more than one CPU, and takes sufficient advantage of multitasking and concurrent execution to provide performance gains over a uniprocessor implementation.

VxWorks SMP Operating System Features

With few exceptions, the SMP and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few functions. A few uniprocessor APIs are not suitable for an SMP system, and they are therefore not supported. Similarly, SMP-specific APIs are not relevant to a uniprocessor system—but default to appropriate uniprocessor behaviors (such as task spinlocks defaulting to task locking), or have no effect.

VxWorks SMP is designed for symmetric target hardware. That is, each CPU has equivalent access to all memory and all devices, except those privately owned by individual CPUs (For example, each CPU might have its own private timers, and for Intel, each CPU has its own local interrupt controller.).

VxWorks SMP can therefore run on targets with multiple single-core processors or with multicore processors, as long as they provide a uniform memory access (UMA) architecture with hardware-managed cache-coherency.

This section provides a brief overview of areas in which VxWorks offers alternate or additional features designed for symmetric multiprocessing. The topics are covered in detail later in this chapter.

Multitasking

SMP changes the conventional uniprocessor paradigm of priority-based preemptive multitasking programming, because it allows true concurrent execution of tasks and handling of interrupts. This is possible because multiple tasks can run on multiple CPUs, while being controlled by a single instance of an operating system.

Uniprocessor multitasking environments are often described as ones in which multiple tasks can run *at the same time*, but the reality is that the CPU only executes one task at a time, switching from one task to the another based on the characteristics of the scheduler and the arrival of interrupts. In an SMP system concurrent execution is a fact and not an illusion.

Scheduling

VxWorks SMP provides a priority-based preemptive scheduler, similar in many ways to the VxWorks uniprocessor (UP) scheduler. However, the VxWorks SMP scheduler differs from the UP scheduler in that it manages the concurrent execution of tasks on different CPUs. The SMP scheduler provides a variety of scheduling policies.

For more information, see *18.12 SMP Scheduling*, p.436.

Mutual Exclusion

Because SMP systems allow for truly concurrent execution, the uniprocessor mechanisms for disabling (masking) interrupts and for suspending task preemption in order to protect critical regions are inappropriate for—and not available in—an SMP operating system. Enforcing interrupt masking or suspending task preemption across all CPUs would defeat the advantages of truly concurrent execution and drag multiprocessing performance down towards the level of a uniprocessor system.

VxWorks SMP therefore provides specialized mechanisms for mutual exclusion between tasks and interrupts executing and being received (respectively) simultaneously on different CPUs. In place of uniprocessor task and interrupt locking functions—such as **taskLock( )** and **intLock( )**—VxWorks SMP provides spinlocks, atomic memory operations, and CPU-specific mutual exclusion facilities.

CPU Affinity

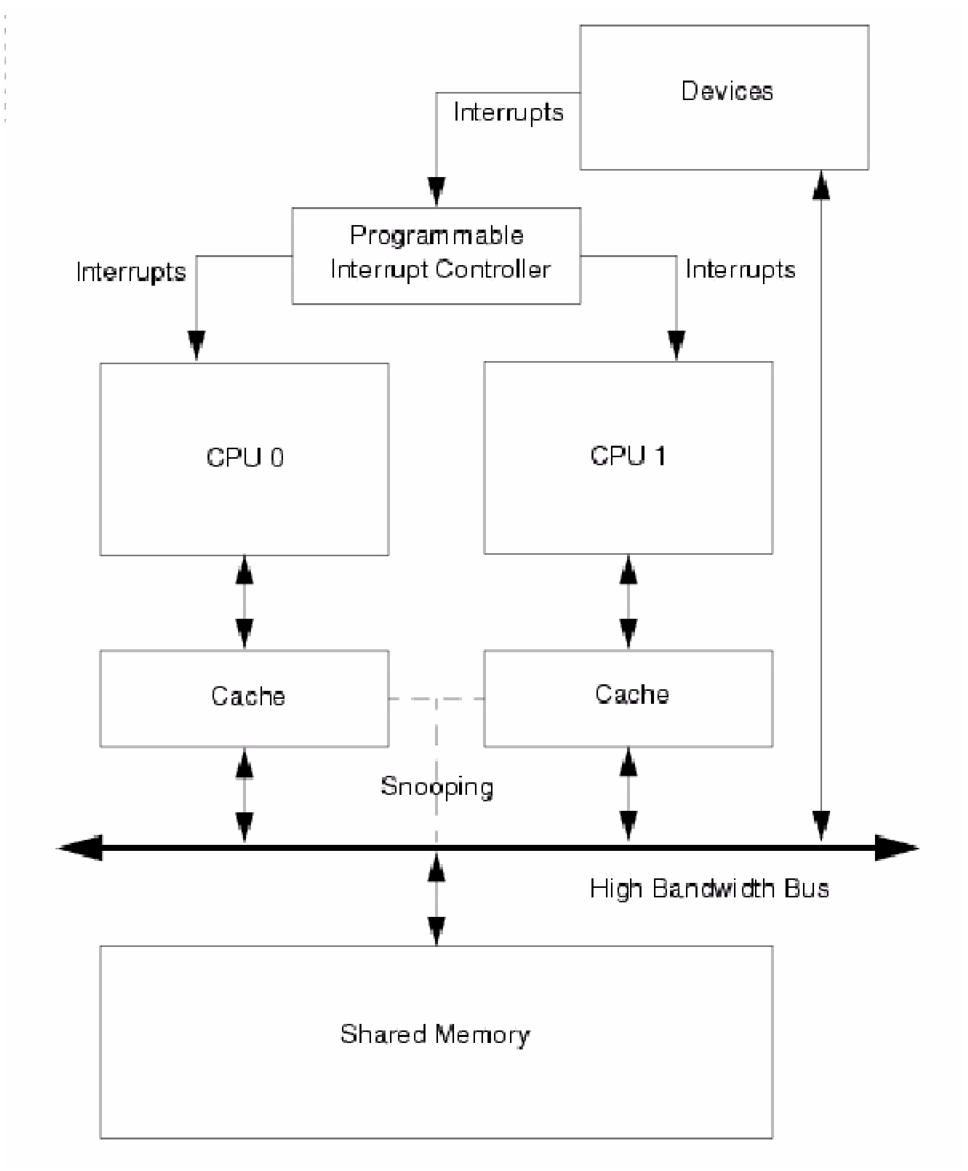
By default, any task can run on any of the CPUs in the system (which generally provides the best load balancing) and interrupts are routed to the VxWorks instance's logical CPU 0. There are instances, however, in which it is useful to assign specific tasks or interrupts to a specific logical CPU. VxWorks SMP provides this capability, which is referred to a as *CPU affinity*.

VxWorks SMP Hardware

The hardware required for use with VxWorks SMP must consist of symmetric multiprocessors—either multicore processors or hardware systems with multiple single CPUs. The processors must be identical, all memory must be shared between the CPUs (none may be local to a CPU), and all devices must be equally accessible from all CPUs.That is, targets for VxWorks SMP must adhere to the uniform memory access (UMA) architecture.

Figure 18-1 illustrates the typical target hardware for a dual CPU SMP system.

Figure 18-1 **SMP Hardware**



Regardless of the number of CPUs (typically 2, 4 or 8) in an SMP system, the important characteristics are the same:

- Each CPU accesses the very same physical memory subsystem; there is no memory local to a CPU. This means it is irrelevant which CPU executes code.
- Each CPU has its own memory management unit that allows concurrent execution of tasks with different virtual memory contexts. For example, CPU 0 can execute a task in RTP 1 while CPU 1 executes a task in RTP 2.
- Each CPU has access to all devices (except those privately owned by individual CPUs (for example, each CPU might have its own private timers, and for Intel, each CPU has its own local interrupt controller). Interrupts from these devices can be routed to any one of the CPUs through a programmable

interrupt controller. This means that it is irrelevant which CPU executes interrupt service routines (ISRs) when handling interrupts.

- Tasks and ISRs can be synchronized across CPUs and mutual exclusion enforced by using spinlocks.
- Bus snooping logic ensures the data caches between CPUs are always coherent. This means that the operating system does not normally need to perform special data cache operations in order to maintain coherent caches. However, this implies that only memory access attributes that allow bus snooping are used in the system. Restrictions in terms of memory access modes allowed in an SMP system, if any, are specific to a hardware architecture.

Comparison of VxWorks SMP and AMP

The features of VxWorks SMP may be highlighted by comparison with the way VxWorks UP instances are used in an asymmetric multiprocessing (AMP) system—with the same target hardware in both cases. The relationship between CPUs and basic uses of memory in SMP and AMP systems are illustrated in Figure 18-2 and Figure 18-3. (The option of using SMP instances in an AMP system is not dealt with in this section.)

Figure 18-2 **SMP System**

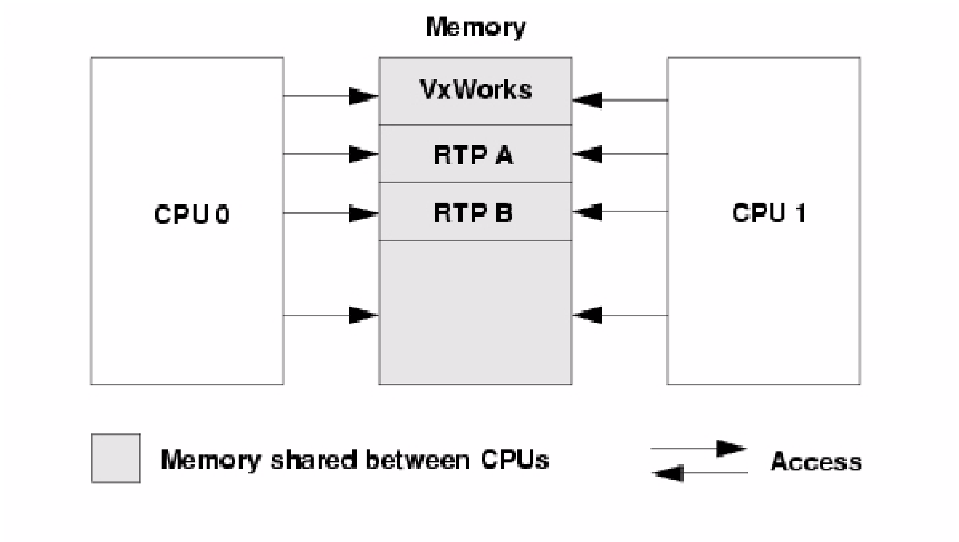
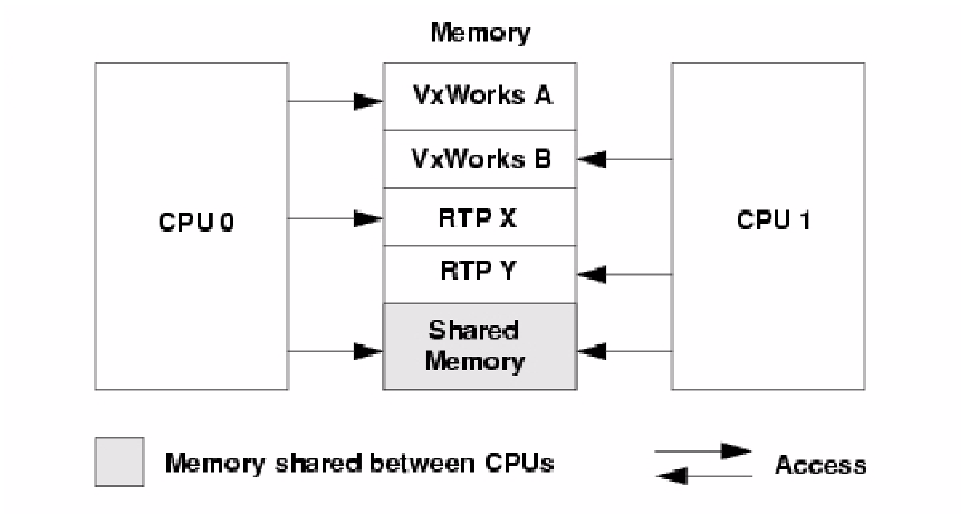


Figure 18-3 **AMP System**



In a purely SMP system the entire physical memory space is shared between the CPUs. This memory space is used to store a single VxWorks SMP image (text, data, bss, heap). It is also used to store any real-time processes (RTPs) that are created during the lifetime of the system. Because both CPUs can potentially read from, write to and execute any memory location, any kernel task or user (RTP) task can be executed by either CPU.

In an AMP configuration there is one VxWorks image in memory for *each* CPU. Each operating system image can only be accessed by the CPU to which it belongs. It is therefore impossible for CPU 1 to execute kernel tasks residing in VxWorks CPU 0's memory, or the reverse. The same situation applies to RTPs. An RTP can only be accessed and executed by the instance of VxWorks from which it was started.

In an AMP system some memory is shared, but typically the sharing is restricted to reading and writing data. For example, for passing messages between two instances of VxWorks. Hardware resources are mostly divided between instances of the operating system, so that coordination between CPUs is only required when accessing shared memory.

With an SMP system, both memory and devices are shared between CPUs, which requires coordination within the operating system to prevent concurrent access to shared resources.

18.2 VxWorks SMP Configuration

Using VxWorks SMP requires selection of both VSB and VIP options.

VSB Configuration

If you use Workbench, create your VSB project and then select the **SMP** option.

If you use **wrtool**, use the **-smp** option with the **prj vsb create** command when you create your VSB project.

For either Workbench or **wrtool**, if you want to use the alternate version of the kernel lock facility (for critical sections of the kernel) that also locks interrupts on the local CPU, select the **INTLOCKED_KERNEL_STATE** option. For more information, see *18.9 Kernel Lock Options*, p.432.

VIP Configuration

Using Workbench or **wrtool**, base your VIP project on the VSB project that you have configured and built with SMP support.

If you use **wrtool**, also use the **-smp** option with the **prj vip create** command when you create the project.

The CPU configuration parameters provide for defining use of the system's CPUs. The idle task configuration parameters allow for configuring the exception stack for the CPU idle task.

Optionally, you can add support for debug versions of spinlocks and a scheduler policy for load balancing a simultaneous multi-threading (SMT) system on a superscalar processor.

⚠ **CAUTION:** VxWorks SMP does not support MMU-less configurations.

CPU Configuration Parameters

The **INCLUDE_KERNEL** component provides several configuration parameters specific to VxWorks SMP. These parameters are as follows:

VX_SMP_NUM_CPUS

Determines the number of CPUs that should be enabled for VxWorks SMP. It must not exceed the maximum number of CPUs defined by the **VX_MAX_SMP_CPUS** macro in the architecture-specific header file. Wind River recommends configuring for the actual number of CPUs that you need (and not more) in order to conserve memory.

⚠ **CAUTION:** Be sure that the number of CPUs allocated for VxWorks SMP is appropriate for the architecture.

ENABLE_ALL_CPUS
Enables all CPUs that have been configured for the system's use with **VX_SMP_NUM_CPUS**. The default is **TRUE**, in which case VxWorks boots with all CPUs enabled and running. The parameter can be set to **FALSE** for debugging purposes, in which case only the VxWorks instance's logical CPU 0 is enabled. The **kernelCpuEnable( )** function can then be used to enable a specific CPU once the system has booted.

VX_ENABLE_CPU_TIMEOUT
The time-out value (in seconds) for the period during which additional cores may be enabled. When **kernelCpuEnable( )** is called, it waits for the time defined by **VX_ENABLE_CPU_TIMEOUT** for the additional core to come up. If **ENABLE_ALL_CPUS** is set to **TRUE**, the value of **VX_ENABLE_CPU_TIMEOUT** is used as the time-out period for enabling all CPUs.

VX_SMP_CPU_EXPLICIT_RESERVE
By default, all CPUs are available for reservation for CPU-affinity tasks (as long as a **vxCpuReserve( )** call has not reserved a CPU or CPUs). The **VX_SMP_CPU_EXPLICIT_RESERVE** parameter can, however, be used to exclude CPUs from the pool that can be chosen arbitrarily for reservation by specifying logical CPU indexes. For example, assigning the string "2 3 7" to this parameter means that logical CPUs 2, 3 and 7 would not be used for arbitrary reservation.

When CPUs are excluded with **VX_SMP_CPU_EXPLICIT_RESERVE**, the only way to reserve any of them is with **vxCpuReserve( )**. For information about task affinity and CPU reservation, see *18.10 CPU Affinity*, p. 432 *18.11 CPU Reservation for CPU-Affinity Tasks*, p. 435.

Idle Task Configuration Parameters

The **INCLUDE_PROTECT_IDLE_TASK_STACK** component provides a set of parameters for configuring the exception stack for the CPU idle task. For information about idle tasks, see *CPU Idle Tasks*, p. 421.

IDLE_TASK_EXCEPTION_STACK_SIZE
Size (in bytes) of the idle tasks' exception stacks.

IDLE_TASK_EXC_STACK_OVERFLOW_SIZE
Size (in bytes) of the overflow protection area adjacent to the idle task's exception stack.

IDLE_TASK_EXC_STACK_UNDERFLOW_SIZE.
Size (in bytes) of the underflow protection area adjacent to the idle task's exception stack.

Debug Versions of Spinlock Components

The **INCLUDE_SPINLOCK_DEBUG** component provides versions of spinlocks that are useful for debugging SMP applications (by default the standard **INCLUDE_SPINLOCK** component is included in VxWorks SMP). For more information, see *Debug Versions of Spinlocks*, p. 426.

If you include debug spinlocks, you may want to include the **INCLUDE_EDR_ERRLOG** component as well so that any spinlock errors are recorded. Without the **INCLUDE_EDR_ERRLOG** component, spinlock errors simply cause the system to reboot. For information about error logging, see *13. Error Detection and Reporting*.

SMP Scheduler Policies

VxWorks SMP provides several scheduling policies. For information about these policies and their configuration, see *18.12 SMP Scheduling*, p. 436.

18.3 VxWorks SMP Boot Process

When VxWorks SMP is configured to boot from physical CPU 0 (the default), the process is essentially the same operation as booting VxWorks UP.

The boot loader is simply responsible for booting the *bootstrap CPU* (in this case physical CPU 0). The boot loader itself has no knowledge of any CPU other than the bootstrap CPU. Once the VxWorks SMP image is loaded on CPU 0 and started, that instance of the operating system enables the other CPUs in the system.

VxWorks SMP can be configured so that the image does not automatically enable any CPUs other than the logical CPU 0. The others can then be enabled interactively or programmatically. For more information in this regard, see *ENABLE_ALL_CPUS*, p. 418).

CAUTION: Boot loaders for VxWorks SMP must not be built with the SMP build option. Boot loaders built with the SMP build option will not function properly.

18.4 Programming for VxWorks SMP

Programming for VxWorks SMP and VxWorks UP is in many respects the same. With few exceptions, the SMP and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few functions.

A few uniprocessor APIs are not suitable for an SMP system, and they are therefore not supported. Similarly, SMP-specific APIs are not relevant to a uniprocessor system—but default to appropriate uniprocessor behaviors (such as task spinlocks defaulting to task locking), or have no effect.

However, because of the nature of SMP systems, SMP programming requires special attention to the mechanisms of mutual exclusion, and to design considerations that allow for full exploitation of the capabilities of a multiprocessing system. Also note that VxWorks SMP maintains an *idle task* for each CPU, and that idle tasks must not be interfered with.

SMP and Mutual Exclusion

The use of mutual exclusion facilities is one of the critical differences between uniprocessor and SMP programming. While some facilities are the same for VxWorks UP and VxWorks SMP, others are necessarily different. In addition, reliance on implicit synchronization techniques—such as relying on task priority instead of explicit locking—do not work in an SMP system (for more information on this topic, see *Implicit Synchronization of Tasks*, p. 452).

Unlike uniprocessor systems, SMP systems allow for truly concurrent execution, in which multiple tasks may execute, and multiple interrupts may be received and serviced, all at the same time. In most cases, the same mechanisms—semaphores, message queues, and so on—can be used in both uniprocessor and SMP systems for mutual exclusion and coordination of tasks (see *7. Intertask and Interprocess Communication*).

However, the specialized uniprocessor mechanisms for disabling (masking) interrupts and for suspending task preemption in order to protect critical regions are inappropriate for—and not available in—an SMP system. This is because they would defeat the advantages of truly concurrent execution by enforcing masking or preemption across all CPUs, and thus drag a multiprocessing system down towards the performance level of uniprocessor system.

The most basic differences for SMP programming therefore have to do with the mechanisms available for mutual exclusion between tasks and interrupts executing and being received (respectively) on different CPUs. In place of uniprocessor task and interrupt locking functions—such as **taskLock( )** and **intLock( )**—VxWorks SMP provides the following facilities:

- spinlocks for tasks and ISRs
- CPU-specific mutual exclusion for tasks and ISRs
- atomic memory operations
- memory barriers

As with the uniprocessor mechanisms used for protecting critical regions, spinlocks and CPU-specific mutual exclusion facilities should only used when they are guaranteed to be in effect for very short periods of time. The appropriate use of these facilities is critical to making an application *SMP-ready* (see *Terminology*, p.412).

Note that both spinlocks and semaphores provide full memory barriers (in addition to the memory barrier macros themselves).

For more information about these topics, see *18.5 Spinlocks for Mutual Exclusion and Synchronization*, p.422, *18.6 CPU-Specific Mutual Exclusion*, p.428, *18.7 Memory Barriers*, p.429, and *18.8 Atomic Memory Operations*, p.431.

CPU Affinity for Interrupts and Tasks

By default, any task can run on any CPUs that is assigned to their instance of VxWorks (which generally provides the best load balancing). Also by default, interrupts are routed to *logical* CPU 0 for that instance of VxWorks. There are cases, however, in which it is useful to assign tasks or interrupts to a specific CPU. VxWorks SMP provides this capability, which is referred to a as *CPU affinity*.

For more information about interrupt and CPU affinity, see *18.10 CPU Affinity*, p.432.

CPU Idle Tasks

VxWorks SMP includes a per-CPU idle task that does not exist in VxWorks UP. The idle task has the lowest priority in the system, below the range permitted for application use (for more information, see *Task Priorities*, p.86). Idle tasks make an SMP system more efficient by providing task context when a CPU enters and exits an idle state. Do not perform any operations that affect the execution of an idle task.

The existence of idle tasks does not affect the ability of a CPU to go to sleep (when power management is enabled) if there is no work to perform.

The **kernelIsCpuIdle( )** and **kernelIsSystemIdle( )** functions provide information about whether a specific CPU is executing an idle task, or whether all CPUs are executing idle tasks (respectively).

For information about configuration options for idle tasks, see *18.2 VxWorks SMP Configuration*, p.417.

WARNING: Do not suspend, stop, change the priority, attempt a task trace, or any similar operations on an idle task. Deleting, suspending, or stopping an idle task causes the system to crash due to an exception in the scheduler. That is, simply do not use the task ID (**tid**) of an idle task as a parameter to any VxWorks function except **taskShow( )**.

RTP Applications

As in VxWorks UP systems, RTP (user mode) applications have a more limited set of mutual exclusion and synchronization mechanisms available to them than kernel code or kernel applications. In VxWorks SMP, they can make use of semaphores and atomic operations, but not spinlocks, memory barriers, or CPU-specific mutual exclusion mechanisms. In addition, the **semExchange( )** function provides for an atomic give and exchange of semaphores.

Optimization for SMP

Using the appropriate facilities in the appropriate manner alone allows an application to execute properly on an SMP system, but does not necessarily take full advantage of the multiprocessing capabilities of the hardware. In order to do so, the design of the application must be geared to exploiting the advantages offered by SMP.

For information in this regard, see *18.15 SMP Performance Optimization*, p. 445.

18.5 Spinlocks for Mutual Exclusion and Synchronization

Spinlocks provide a facility for short-term mutual exclusion and synchronization in an SMP system. They must be explicitly taken and released.

While semaphores can also be used for mutual exclusion and synchronization, spinlocks are designed for use in situations comparable to those in which **taskLock( )** and **intLock( )** are used in VxWorks UP systems. Semaphores should be used in an SMP system for the same purposes as in a UP system. (Note that VxWorks provides scalable and inline variants of semaphore functions that are particularly useful for SMP systems. For more information, see *Scalable and Inline Semaphore Take and Give Kernel Functions*, p. 119).

The basic implementation of VxWorks spinlocks provides algorithms that ensure that they are *fair*, meaning that they are deterministic during the time between the request and take, and they operate as close to FIFO order as possible. Two variant sets of spinlocks are also provided: one that does not guarantee fairness or determinism, but that provides higher performance; and another that provides debug information. (For more information, see *ISR-Callable Spinlocks*, p. 423 and *Debug Versions of Spinlocks*, p. 426).

For information about why uniprocessor mechanisms are not supported on VxWorks SMP for interrupt locking and suspension of task preemption, and about the alternatives used for SMP, see *Synchronization and Mutual Exclusion Facilities*, p. 452, *Interrupt Locking: intLock( ) and intUnlock( )*, p. 454, *Task Locking: taskLock( )*

and taskUnlock( ), p.455, and *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p.455.)

NOTE: Spinlocks are not available to RTP (user-mode) applications.

Spinlocks as Full Memory Barriers

VxWorks spinlocks operate as full memory barriers between acquisition and release (as do semaphores). A full memory barrier forces both read and write memory access operations to be performed in strict order. The process of updating data structures is therefore fully completed between the time a spinlock is acquired and released.

Types of Spinlocks

VxWorks SMP provides the following two types of spinlocks:

- *ISR-callable spinlocks*, which are used to address contention between ISRs—or between a task and other tasks and ISRs. They disable (mask) interrupts on the local CPU. When called by tasks they suspend task preemption on the local CPU as well.
- *Task-only spinlocks*, which are used to address contention between tasks (and not ISRs). They suspend task preemption on the local CPU.

The *local CPU* is the one on which the spinlock call is performed. For detailed information about spinlocks, see *ISR-Callable Spinlocks*, p.423 and *Task-Only Spinlocks*, p.425.

Spinlock Behavior and Usage Guidelines

Unlike the behavior associated with semaphores, a task that attempts to take a spinlock that is already held by another task does not pend; instead it continues executing, simply spinning in a tight loop waiting for the spinlock to be freed.

The terms *spinning* and *busy waiting*—which are both used to describe this activity—provide insight into both the advantages and disadvantages of spinlocks. Because a task (or ISR) continues execution while attempting to take a spinlock, the overhead of rescheduling and context switching can be avoided (which is not the case with a semaphore). On the other hand, spinning does no useful work, and ties up one or more of the CPUs.

Spinlocks should therefore only be used when they are likely to be efficient; that is, when they are going to be held for very short periods of time (as with **taskLock( )** and **intLock( )** in a uniprocessor system). If a spinlock is held for a long period of time, the drawbacks are similar to **intLock( )** and **taskLock( )** being held for a long time in VxWorks UP—increased interrupt and task latency.

Acquisition of a spinlock on one CPU does not affect the processing of interrupts or scheduling of tasks on other CPUs. Tasks cannot be deleted while they hold a spinlock.

For detailed cautionary information about spinlock use, see *Caveats With Regard to Spinlock Use*, p.425 and *Functions Restricted by Spinlock Use*, p.426.

ISR-Callable Spinlocks

Spinlocks that are used to address contention between ISRs—or between a task and other tasks and ISRs—are referred to as *ISR-callable spinlocks*. These spinlocks can be acquired by both tasks and ISRs. VxWorks provides two variants of ISR-callable spinlocks: deterministic and non-deterministic.

Note that when used in a uniprocessor system, ISR-callable spinlocks have the same behavior as the interrupt locking functions **intLock( )** and **intUnlock( )**.

Deterministic ISR-Callable Spinlocks

Deterministic ISR-callable spinlocks are both fair and deterministic: the spinlock is always given to the first one in a queue of requesters, and the time it takes for a spinlock to be taken is bounded. These spinlocks disable (mask) interrupts on the local CPU, which prevents the caller from being preempted while it holds the spinlock (which could otherwise lead to a livelock—a situation in which the caller would spin without ever acquiring the spinlock). If a *task* (as opposed to an ISR) acquires an ISR-callable spinlock, task preemption is also blocked on the local CPU while that task holds the spinlock. This allows the task to execute the critical section that the spinlock is protecting. Interrupts and tasks on other CPUs are not affected.

The standard functions used for ISR-callable spinlocks are listed in Table 18-1. For detailed information, see the **spinLockLib** entry in the VxWorks API references.

Table 18-1 **ISR-Callable Spinlock Functions**

Function	Description
spinLockIsrInit( )	Initializes an ISR-callable spinlock.
spinLockIsrTake( )	Acquires an ISR-callable spinlock.
spinLockIsrGive( )	Relinquishes ownership of an ISR-callable spinlock.

NOTE: The **SPIN_LOCK_ISR_DECL** macro can be used to statically initialize ISR-callable spinlocks. It should be used for code that is intended be portable between 32-bit and 64-bit VxWorks. Standard static array initialization is not portable.

Non-Deterministic ISR-Callable Spinlocks

Non-deterministic variant of ISR-callable spinlocks provides higher performance, but they are not guaranteed to be fair or deterministic when multiple CPUs attempt to take a spinlock simultaneously.

Unlike deterministic spinlocks, non-deterministic spinlocks do not guarantee that the spinlock is always given to the first requestor. Indeed, a CPU may never get a spinlock as it is always possible for another CPU to get the spinlock before it does. And they are non-deterministic because it is not possible to tell how long it will take for them to be taken by a given requestor. These spinlocks do, however, have good interrupt latency properties, as interrupts are not locked while a CPU waits to acquire the lock.

The non-deterministic variants are listed in Table 18-2. For detailed information, see the **spinLockIsrNDLib** entry in the VxWorks API references.

Table 18-2 **Non-Deterministic ISR-Callable Spinlock Functions**

Function	Description
spinLockIsrNdInit( )	Initializes a non-deterministic ISR-callable spinlock.
spinLockIsrNdTake( )	Acquires a non-deterministic ISR-callable spinlock.
spinLockIsrNdGive( )	Relinquishes ownership of a non-deterministic ISR-callable spinlock.

Task-Only Spinlocks

Spinlocks that are used to address contention between tasks alone (and not ISRs) are called *task-only spinlocks*. These spinlocks disable task preemption on the local CPU while the caller holds the lock (which could otherwise lead to a livelock situation). This prevents the caller from being preempted by other tasks and allows it to execute the critical section that the lock is protecting. Interrupts are not disabled and task preemption on other CPUs is not affected. The functions used for task-only spinlocks are listed in Table 18-3.

For VxWorks UP, task-only spinlocks are implemented with the same behavior as the task locking functions **taskLock( )** and **taskUnlock( )**.

Table 18-3 **Task-Only Spinlock Functions**

Function	Description
spinLockTaskInit( )	Initializes a task-only spinlock.
spinLockTaskTake( )	Acquires a task-only spinlock.
spinLockTaskGive( )	Relinquishes ownership of a task-only spinlock.

NOTE: The **SPIN_LOCK_TASK_DECL** macro can be used to statically initialize task-only spinlocks. It should be used for code that is intended be portable between 32-bit and 64-bit VxWorks. Standard static array initialization is not portable.

Caveats With Regard to Spinlock Use

Because of the concurrency of execution inherent in SMP systems, spinlocks should be used with care. The following prescriptions should be adhered to avoid problems with spinlocks:

- A spinlock should only be held for a short and deterministic period of time.
- A task or ISR must not take more than one spinlock at a time. Livelocks may result when an entity that already holds a spinlock takes another spinlock. Livelocks are similar to the deadlocks that occur with semaphore use. With spinlocks, however, the entity does not pend or block; it spins without ever acquiring the spinlock and the CPU appears to be hung. Because interrupts are masked or task preemption is disabled, the state cannot be remedied.
- A task or ISR must not take a spinlock that it already holds. That is, recursive takes of a spinlock should not be made. A livelock will occur.

- In order to prevent a task or ISR from entering a kernel critical region while it already holds a spinlock—and cause the system to enter a livelock state—a task or ISR must not call specified functions while it holds a spinlock. The VxWorks SMP kernel itself uses spinlocks to protect its critical regions. For information about these functions, see *Functions Restricted by Spinlock Use*, p.426.

A debug version of spinlocks can be used to catch these problems. For information, see *Debug Versions of Spinlocks*, p.426.

Debug Versions of Spinlocks

The debug version of VxWorks spinlocks (provided with the **INCLUDE_SPINLOCK_DEBUG** component) is designed for use while developing applications that use spinlocks. It allows for catching violations of guidelines for appropriate spinlock use (for information in this regard, see *Caveats With Regard to Spinlock Use*, p.425).

Note that if the **INCLUDE_EDR_ERRLOG** component is not included in the system as well, the debug version of spinlocks will simply reboot the system without any information when an error is encountered. With **INCLUDE_EDR_ERRLOG** component, the messages are logged.

The following is a list of checks that are performed by the debug version of spinlocks:

- task-only take with spinLockTaskTake( )
  - Calling from an ISR context results in an error.
  - Recursive taking of a spinlock results in an error.
  - Nested taking of spinlocks results in an error.
- task-only give with spinLockTaskGive( )
  - Calling from an ISR context results in an error.
  - Attempting to give up a spinlock without first acquiring a spinlock results in an error.
- ISR-only take with spinLockIsrTake( )
  - Recursive taking of a spinlock results in an error.
  - Nested taking of spinlocks result in an error.
- ISR-only give with spinLockIsrGive( )
  - Attempting to give up a spinlock without first acquiring a spinlock results in an error.

Errors are handled by the error detection and reporting facility, in the form of a fatal kernel error with an appropriate error string (for information about the error detection and reporting facility, see *13. Error Detection and Reporting*).

Functions Restricted by Spinlock Use

Certain functions should not be called while the calling entity (task or ISR) holds a spinlock. This restriction serves to prevent a task or ISR from entering a kernel critical region while it already holds a spinlock—and cause the system to enter a livelock state (for more information, see *Caveats With Regard to Spinlock Use*, p.425). The function restriction also apply to **intCpuLock( )** (for more information about this function see *CPU-Specific Mutual Exclusion for Interrupts*, **p.428**). This restriction applies because the kernel requires interrupts to be enabled to implement its multi-CPU scheduling algorithm.

It is outside the scope of this document to list all the VxWorks spinlock restricted functions. However, generally speaking these are functions related to the creation, destruction and manipulation of kernel objects (semaphores, tasks, message queues, and so on) as well as any function that can cause a scheduling event.

While the restriction imposed by spinlock use may seem to be a hindrance, it really should not be. Spinlocks are meant for very fast synchronization between processors. Holding a spinlock and attempting to perform notable amounts of work, including calling into the kernel, results in poor performance on an SMP system, because either task preemption or interrupts, or both, are disabled when a CPU owns a spinlock.

Table 18-4 identifies some of the functions restricted by spinlock and CPU lock use.

Table 18-4 **Functions Restricted by Spinlock and CPU Lock Use**

Library	Functions
taskLib	**taskExit( )**, **taskDelete( )**, **taskDeleteForce( )**, **taskInitExcStk( )**, **taskUnsafe( )**, **exit( )**, **taskSuspend( )**, **taskResume( )**, **taskPrioritySet( )**, **taskDelay( )**, **taskStackAllot( )**, **taskRestart( )**, **taskCpuLock( )**, **taskCpuUnlock( )**, **taskCreateLibInit( )**, **taskCreate( )**, **taskActivate( )**, **taskCpuAffinitySet( )**, **taskCpuAffinityGet( )**, **taskSpawn( )**, **taskInit( )**
msgQLib	**msgQCreate( )**, **msgQDelete( )**, **msgQSend( )**, **msgQReceive( )**, **msgQInitialize( )**, **msgQNumMsgs( )**,
msgQEvLib	**msgQEvStart( )**, **msgQEvStop( )**
semLib	**semTake( )**, **semGive( )**, **semFlush( )**, **semDelete( )**
semBLib	**semBInitialize( )**, **semBCreate( )**
semCLib	**semCInitialize( )**, **semCCreate( )**
semMLib	**semMInitialize( )**, **semMGiveForce( )**, **semMCreate( )**
semEvLib	**semEvStart( )**, **semEvStop( )**
wdLib	**wdCreate( )**, **wdDelete( )**, **wdinitialise( )**, **wdStart( )**, **wdCancel( )**
kernelLib	**kernelTimeSlice( )**, **kernelCpuEnable( )**
intLib	**intDisconnect( )**
intArchLib	**intConnect( )**, **intHandlerCreate( )**, **intVecTableWriteProtect( )**
eventLib	**eventSend( )**, **eventReceive( )**

18.6 CPU-Specific Mutual Exclusion

VxWorks SMP provides facilities for CPU-specific mutual exclusion, that is for mutual exclusion operations whose scope is entirely restricted to the CPU on which the call is made (the *local* CPU). These facilities are designed to facilitate porting uniprocessor code to an SMP system.

CPU-Specific Mutual Exclusion for Interrupts

CPU-specific mutual exclusion for interrupts allows for disabling (masking) interrupts on the CPU on which the calling task or ISR is running. For example if task A, running on *logical* CPU 0, performs a local CPU interrupt lock operation, no interrupts can be processed by CPU 0 until the lock is released by task A.

Execution of interrupts on other CPUs in the SMP system is not affected. In order to be an effective means of mutual exclusion, therefore, all tasks and ISRs that should participate in the mutual exclusion scenario should have CPU affinity set for the local CPU (for information, see *Task CPU Affinity*, p.433).

Note that some functions should not be used if the calling task or ISR has locked interrupts on the local CPU—similar to the case of holding spinlocks (see *Caveats With Regard to Spinlock Use*, p.425). The restricted functions are described in *Functions Restricted by Spinlock Use*, p.426.

The functions listed in Table 18-5 are used for disabling and enabling interrupts on the local CPU.

Note that in a uniprocessor system they default to the behavior of **intLock( )** and **intUnlock( )**.

Table 18-5 **CPU-Specific Mutual Exclusion Functions for Interrupts**

Function	Description
intCpuLock( )	Disables interrupts on the CPU on which the calling task or ISR is running.
intCpuUnlock( )	Enables interrupts on the CPU on which the calling task or ISR is running.

For more information about these functions, see the **intLib** entry in the VxWorks API references.

CPU-Specific Mutual Exclusion for Tasks

CPU-specific mutual exclusion for tasks allows for suspending task preemption on the CPU on which the calling task is running. That is, it provides for local CPU task locking, and effectively prevents any other task from running on the local CPU. For example, task A running on CPU 0 can perform a local CPU task lock operation so that no other task can run on CPU 0 until it releases the lock or makes a blocking call.

The calling task is also prevented from migrating to another CPU until the lock is released.

Execution on other CPUs in the SMP system is not affected. In order to be an effective means of mutual exclusion, therefore, all tasks that should participate in the mutual exclusion scenario should have CPU affinity set for the local CPU (for information, see *Task CPU Affinity*, p.433).

The functions listed in Table 18-6 are used for suspending and resuming task preemption on the local CPU.

Note that in a uniprocessor system they default to the behavior of **taskLock( )** and **taskUnlock( )**.

Table 18-6 **CPU-Specific Mutual Exclusion Functions for Tasks**

Function	Description
taskCpuLock( )	Disables task preemption for the CPU on which the calling task is running.
taskCpuUnlock( )	Enables context task switching on the CPU on which the calling task is running.

For more information about these functions, see the **taskLib** entry in the VxWorks API references.

18.7 Memory Barriers

VxWorks provides a set of memory barrier operations which provide a way to guarantee the ordering of operations between cooperating CPUs.

In modern multiprocessing architectures, individual CPUs can reorder both read and write operations in order to improve overall system efficiency. From the perspective of a single CPU in the system, this reordering is completely transparent because the CPU ensures that any read operation gets data that was previously written, regardless of the order in which the read and write operations are actually committed to system memory. The reordering occurs in the background, and is never visible to the programmer.

In an multiprocessor system, an individual CPU can execute a series of write operations to memory, and these write operations can be queued between the CPU and system memory. The CPU is allowed to commit these queued operations to system memory in any order, regardless of the order in which the operations arrive in the CPU's *write queue*. Similarly, a CPU is free to issue more than one read operation in parallel, whether as the result of speculative execution, or because the program has requested more than one independent read operation.

Because of this reordering, two tasks that share data should never assume that the order in which an operation is performed on one CPU will be the same as the order in which the operations are written to or read from memory. A classic example of this ordering problem involves two CPUs, in which one CPU prepares an item of work to be performed, and then sets a boolean flag to announce the availability of the work unit to a second CPU that is waiting for it. The code in this case would look like the following:

```
/* CPU 0 - announce the availability of work */

pWork = &work_item;  /* store pointer to work item to be performed */
```

```
workAvailable = 1;

/* CPU 1 - wait for work to be performed */

while (!workAvailable);
doWork (pWork);    /* error - pWork might not be visible to this CPU yet */
```

It is very likely that the **pWork** pointer used by CPU 1 will contain incorrect data because CPU 0 reorders its write operations to system memory, which causes CPU 1 to observe the change to the **workAvailable** variable before the value of the **pWork** variable has been updated. In a case like this, the likely result is a system crash due to de-referencing an invalid pointer.

To solve the memory ordering problem, VxWorks provides a set of memory barrier operations. The sole purpose of memory barrier operations is to provide a way to guarantee the ordering of operations between cooperating CPUs. Memory barriers fall into three general classes:

- read memory barrier
- write memory barrier
- full (read/write) memory barrier

NOTE: VxWorks SMP provides a set of synchronization primitives to protect access to shared resources. These primitives include semaphores, message queues, and spinlocks. These primitives include full memory barrier functionality. Additional memory barrier operations are not required with these facilities to protect shared resources.

Read Memory Barrier

The **VX_MEM_BARRIER_R( )** macro provides a read memory barrier. **VX_MEM_BARRIER_R( )** enforces ordering between all of the read operations that have occurred prior to the barrier, and all of the read operations that occur after the barrier. Without this barrier, a CPU is free to reorder its pending read operations in any way that does not affect program correctness from a uniprocessor perspective. For example, a CPU is free to reorder the following independent reads:

```
a = *pAvalue;    /* read may occur _after_ read of *pBvalue */
b = *pBvalue;    /* read may occur _before read of *pAValue */
```

By inserting a memory barrier between the read operations, you can guarantee that the reads occur in the appropriate order:

```
a = *pAvalue;                   /* will occur before read of *pBvalue */
VX_MEM_BARRIER_R();
b = *pBvalue;                   /* will occur after read of *pAvalue */
```

While **VX_MEM_BARRIER_R( )** can ensure that the read operations occur in the correct order, this guarantee is not helpful unless the writer of the shared data also ensures that the writes of the shared data also occur in the correct order. For this reason, the **VX_MEM_BARRIER_R( )** and **VX_MEM_BARRIER_W( )** macros should always be used together.

Write Memory Barrier

The **VX_MEM_BARRIER_W( )** macro provides a write memory barrier. **VX_MEM_BARRIER_W( )** enforces the ordering between all of the write operations that have occurred prior to the barrier, and all of the write operations that occur after the barrier. The following code fragment is taken from a preceding example, but modified to take advantage of a memory barrier:

```
pWork = &work_item;
VX_MEM_BARRIER_W();
workAvailable = 1;
```

Inserting a barrier between the update of ***pWork** and the update of **workAvailable** ensures that the value of **workAvailable** in system memory is updated after the value of **pWork** has been updated in system memory. Note that **VX_MEM_BARRIER_W( )** does not actually force the writing of these values to system memory. Instead, it merely enforces the order in which these values are written. Note that **VX_MEM_BARRIER_W( )** should always be used with **VX_MEM_BARRIER_R( )** or **VX_MEM_BARRIER_RW( )**.

Read/Write Memory Barrier

The **VX_MEM_BARRIER_RW( )** macro provides a read/write memory barrier. This is also referred to as a *full fence* memory barrier. **VX_MEM_BARRIER_RW( )** combines the effects of both the **VX_MEM_BARRIER_R( )** and **VX_MEM_BARRIER_W( )** operations into a single primitive. On some systems, **VX_MEM_BARRIER_RW( )** may be substantially more expensive than either **VX_MEM_BARRIER_R( )** or **VX_MEM_BARRIER_W( )**. Unless both read and write ordering is required, Wind River does not recommend the use of **VX_MEM_BARRIER_RW( )**.

18.8 Atomic Memory Operations

Atomic operations make use of CPU support for atomically accessing memory. They combine a set of (architecture-specific) operations into what is effectively a single operation that cannot be interrupted by any other operation on the memory location in question. Atomic operations thereby provide mutual exclusion for a simple set of operations (such as incrementing and decrementing variables).

Atomic operations can be useful as a simpler alternative to spinlocks, such as for updating a single data element. For example, you can update the *next* pointer in a singly-linked list from NULL to non-NULL (without interrupts locked) using an atomic operation, which allows you to create lock-less algorithms.

Because the atomic operations are performed on a memory location supplied by the caller, users must ensure the location has memory access attributes and an alignment that allows atomic memory access—otherwise an access exception will occur. Restrictions, if any, are specific to the CPU architecture. For more information, see *Memory-Access Attributes*, p.458.

The **vxAtomicLib** library provides a number of functions that perform atomic operations. They are described in Table 18-7. Note that the library also provides inline versions of these functions—for example, **vxAtomicAdd_inline( )**.

Atomic operation functions are available in user space (for RTP applications) as well as in the kernel.

Table 18-7 **Atomic Memory Operation Functions**

Function	Description
vxAtomicAdd( )	Adds two values atomically.
vxAtomicSub( )	Subtracts one value from another atomically.
vxAtomicInc( )	Increments a value atomically.
vxAtomicDec( )	Decrements a value atomically.
vxAtomicOr( )	Performs a bitwise OR operation on two values atomically.
vxAtomicXor( )	Performs a bitwise XOR operation on two values atomically.
vxAtomicAnd( )	Performs a bitwise AND operation on two values atomically.
vxAtomicNand( )	Performs a bitwise NAND operation on two values atomically.
vxAtomicSet( )	Sets one value to another atomically.
vxAtomicClear( )	Clears a value atomically.
vxCas( )	Performs an atomic compare-and-swap of two values atomically.

18.9 Kernel Lock Options

The kernel lock facility is used to protect critical sections of the kernel. It is private to VxWorks, and must not to be manipulated by users.

By default, it does not lock interrupts on the local CPU when it is taken, which minimizes interrupt latency. The VSB option **INTLOCKED_KERNEL_STATE** provides an alternate version of the kernel lock that does lock interrupts on the local CPU when it is taken. This increases interrupt latency, but addresses the worst case scenario in which the kernel might be interrupted (perhaps serially) for an indeterminate amount of time before another CPU could take the kernel lock and enter a kernel critical section.

18.10 CPU Affinity

VxWorks SMP provides facilities for *CPU affinity*; that is, for assigning specific interrupts or tasks to specific CPUs.

Task CPU Affinity

VxWorks SMP provides the ability to assign tasks to a specific CPU, after which the scheduler ensures the tasks are only executed on that CPU. This assignment is referred to as *task CPU affinity*. (CPUs can also be reserved for their affinity tasks; for information in this regard, see *18.11 CPU Reservation for CPU-Affinity Tasks*, p.435.)

While the default SMP operation—in which any task can run on any CPU—often provides the best overall load balancing, there are cases in which assigning a specific set of tasks to a specific CPU can be useful. For example, if a CPU is dedicated to signal processing and does no other work, the cache remains filled with the code and data required for that activity. This saves the cost of moving to another CPU—which is incurred even within single piece of silicon, as the L1 cache is bound to a single CPU, and the L1 must be refilled with new text and data if the task migrates to a different CPU.

Another example is a case in which profiling an application reveals that some of its tasks are frequently contending for the same spinlock, and a fair amount of execution time is wasted waiting for a spinlock to become available. Overall performance could be improved by setting task CPU affinity such that all tasks involved in spinlock contention run on the same CPU. This would free up more time other CPUs for other tasks.

Setting Task CPU Affinity

A task inherits the CPU affinity of the parent task, if the parent has CPU affinity set. A task created or initialized by any of the following functions inherits the CPU affinity of the calling task: **taskSpawn( )**, **taskCreate( )**, **taskInit( )**, **taskOpen( )**, **taskInitExcStk( )**, and **rtpSpawn( )**.

Note that for RTP tasks, the **RTP_CPU_AFFINITY_NONE** option for **rtpSpawn( )** can be used to create an RTP in which tasks have no CPU affinity, despite the fact that the RTP's parent task itself may have had one.

A task can set its own CPU affinity or the CPU affinity of another task by calling **taskCpuAffinitySet( )**. The **taskLib** library provides functions for managing task CPU affinity. They are described in Table 18-8.

Table 18-8 **Task CPU Affinity Functions**

Function	Description
taskCpuAffinitySet( )	Sets the CPU affinity for a task.
taskCpuAffinityGet( )	Returns the CPU affinity for a task.

The function **taskCpuAffinitySet( )** takes a CPU set variable (of type **cpuset_t**) to identify the logical CPU to which the task should be assigned. Similarly, the **taskCpuAffinityGet( )** function takes a pointer to a **cpuset_t** variable for the purpose of recording the CPU affinity for a given task.

For both functions, the **CPUSET_ZERO( )** macro must be used to clear the **cpuset_t** variable before the call is made.

To set CPU affinity, first use the **CPUSET_ZERO( )** macro to clear the **cpuset_t** variable, then use the **CPUSET_SET( )** macro to specify the CPU, and finally call **taskCpuAffinitySet( )**.

To remove task CPU affinity, use the **CPUSET_ZERO( )** macro to clear the **cpuset_t** variable, and then make the **taskCpuAffinitySet( )** call again.

For more information about using these functions and macros see *Task CPU Affinity Examples*, p.434 and *CPU Set Variables and Macros*, p.442

Task CPU Affinity Examples

The following sample code illustrates the sequence to set the affinity of a newly created task to CPU 1.

```
STATUS affinitySetExample (void)
   {
   cpuset_t affinity;
   int tid;

   /* Create the task but only activate it after setting its affinity */
   tid = taskCreate ("myCpu1Task", 100, 0, 5000, printf,
                     (int) "myCpu1Task executed on CPU 1 !", 0, 0, 0,
                     0, 0, 0, 0, 0, 0);

   if (tid == NULL)
       return (ERROR);

   /* Clear the affinity CPU set and set index for CPU 1 */
   CPUSET_ZERO (affinity);
   CPUSET_SET  (affinity, 1);

   if (taskCpuAffinitySet (tid, affinity) == ERROR)
       {
       /* Either CPUs are not enabled or we are in UP mode */
       taskDelete (tid);
       return (ERROR);
       }

   /* Now let the task run on CPU 1 */
   taskActivate (tid);

   return (OK);
   }
```

The next example shows how a task can remove its affinity to a CPU:

```
   {
   cpuset_t affinity;

   CPUSET_ZERO (affinity);

   taskCpuAffinitySet (0, affinity);
   }
```

Interrupt CPU Affinity

SMP hardware requires programmable interrupt controller devices (for more information see *VxWorks SMP Hardware*, p.414). VxWorks SMP makes use of this hardware to allow assignment interrupts to a specific CPU. By default, interrupts are routed to the VxWorks instance's logical CPU 0.

Interrupt CPU affinity can be useful for load balancing (for example, if there is too much total interrupt traffic for one CPU to handle). It can also be used as an aid in migrating code from VxWorks UP (for more information, see *Interrupt Locking: intLock( ) and intUnlock( )*, p.454).

Runtime assignment of interrupts to a specific CPU occurs at boot time, when the system reads interrupt configuration information from the BSP. The interrupt controller then receives a command directing that a given interrupt be routed to a specific CPU. For information about the mechanism involved, see the *VxWorks Device Driver Developer's Guide*.

18.11 CPU Reservation for CPU-Affinity Tasks

VxWorks SMP provides facilities for reserving CPUs for those tasks that have been assigned *CPU affinity* to them. This prevents the dedicated tasks from being preempted by other tasks in the system, and thus improves their performance.

This facility can, for example, be used to off load networking-related tasks to a dedicated CPU for faster response times.

For information about CPU affinity, see *18.10 CPU Affinity*, p. 432.

CPU Reservation Functions

The **vxCpuLib** library provides functions for managing CPU reservation. They are described in Table 18-9.

Table 18-9 **CPU Reservation Functions**

Function	Description
vxCpuReservedGet( )	Get the set of reserved CPUs.
vxCpuReserve( )	Reserve one or more CPUs.
vxCpuUnreserve( )	Unreserve one or more CPUs.

By default, all CPUs are available for reservation when one or more CPUs are not specified with **vxCpuReserve( )**. The **VX_SMP_CPU_EXPLICIT_RESERVE** configuration parameter can, however, be used to exclude CPUs from the pool of CPUs that can be chosen arbitrarily for reservation. For more information, see *VX_SMP_CPU_EXPLICIT_RESERVE*, p. 419.

There is no required order for calling **vxCpuReserve( )** and **taskCpuAffinitySet( )**. A **taskCpuAffinitySet( )** call sets the affinity of a task to a specific CPU, which prevents the task from migrating to another CPU. A **vxCpuReserve( )** restricts a CPU to only those tasks that have an affinity with it. One call may be made before the other, as suits the needs of the application.

Note that if a task with CPU-affinity creates another task, the new task inherits its parent's CPU-affinity. If the CPU has been reserved for tasks with affinity, the children of those tasks will only run on that CPU. For information about CPU affinity, see *18.10 CPU Affinity*, p. 432.

For more information, see the **vxCpuLib** API reference.

CPU Reservation and Task Affinity Examples

The following code fragment illustrates how to reserve an arbitrary CPU and set the CPU affinity of a task to execute on the reserved CPU:

```
void myRtn (void)
    {
```

```
    cpuset_t    cpuSet;         /* Input argument to vxCpuReserve */
    cpuset_t    resCpuSet;      /* Return argument from vxCpuReserve */

    /* Passing an empty cpuset as input reserves an arbitrary CPU */

    CPUSET_ZERO (cpuSet);

    if (vxCpuReserve (cpuSet, &resCpuSet == OK))
        {
        /* set affinity for current task */

        if (taskCpuAffinitySet (0, resCpuSet) != OK)
        /* handle error */
        }
    else
        {
        /* handle error */
        }
    }
```

The next code fragment illustrates how to reserve one or more specific CPUs and set the CPU affinity for several tasks:

```
void myRtn (void)
    {
    extern int  tids[3];                /* some task IDs */
    int         cpuIx[] = {1,2,4};      /* CPU indices to reserve */
    cpuset_t    cpuSet;
    cpuset_t    tmpCpuSet;
    int         i;

    /* Initialize cpuSet with the desired CPU indices */

    CPUSET_ZERO (cpuSet);
    CPUSET_SET (cpuSet, cpuIx[0]);
    CPUSET_SET (cpuSet, cpuIx[1]);
    CPUSET_SET (cpuSet, cpuIx[2]);

    /* Reserve the specified CPUs */

    if (vxCpuReserve (cpuSet, NULL) == OK)
        {
        for (i = 0; i < 3; i++)
            {
            tmpCpuSet = CPUSET_FIRST_SET (cpuSet);

            if (taskCpuAffinitySet (tids[i], tmpCpuSet) != OK)
            /* handle error */

            CPUSET_SUB (cpuSet, tmpCpuSet);
            }
        }
    else
        {
        /* handle error */
        }
    }
```

18.12 SMP Scheduling

Unlike VxWorks uniprocessor (UP) scheduling, VxWorks SMP scheduling allocates tasks over multiple CPUs. In most respects, SMP scheduling is the same, or similar to, UP scheduling. In both cases the goal is to execute the highest-priority tasks first. The SMP scheduler provides a variety of scheduling policies: default, FETP, and SMT. It also supports RTP time partition scheduling.

Similarities Between UP and SMP Scheduling

Similarities between the two types of scheduling include:

- Both include priority-based preemptive scheduling and, optionally, round-robin scheduling.
- Scheduling decisions are performed whenever either of the two occur:
  - The Wind kernel exits a critical section with **windExit( )**.
  - An ISR completes execution with **intExit( )**.
- There is no task that does the scheduling; the scheduler still invoked when a task or ISR exits a kernel critical section. The scheduling decision is made when the ready queues are updated.
- Latency issues are similar.

Differences Between UP and SMP Scheduling

Table 18-10 shows some of the differences between VxWorks UP scheduling and VxWorks SMP scheduling:

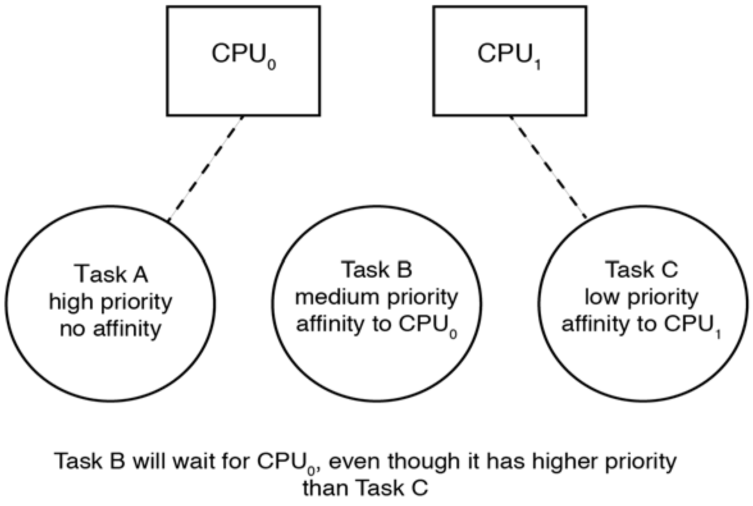
Table 18-10 **Differences Between VxWorks UP and SMP Scheduling**

VxWorks UP Scheduling	VxWorks SMP Scheduling
All tasks are scheduled over a single CPU.	Tasks are split between multiple CPUs (one task per CPU at a time).
The highest-priority task that is ready to run gets the CPU.	The *N* highest-priority tasks that are ready to run will always be running (using best-effort scheduling).
CPU affinity not applicable.	Includes CPU affinity (see *18.10 CPU Affinity*, p.432).
Very efficient, because the scheduler only compares two memory variables: **taskIdCurrent** and the head of the ready queue.	More costly and less deterministic, because of the need to send an IPI to force a task switch.
Some “unfairness” in task allocation due to “clock jitter”.	Additional “unfairness” associated with CPU affinity (see below).

An important part of SMP scheduler functionality is the optional use of *CPU affinity*. With CPU affinity, a task may be assigned to a particular CPU. (See *18.10 CPU Affinity*, p.432.) CPU affinity has many benefits, but it may sometimes produce unexpected scheduling results.

For example, a high-priority task without CPU affinity will not be moved to another CPU to make space for a medium-priority task with CPU affinity, even if the other CPU executes a task that has lower priority than the task with CPU affinity. (See Figure 18-4.) This ensures that the highest-priority task executes without the overhead (for example, context save and restore, cold cache, TLB) incurred when moved to another core.

Figure 18-4 **SMP CPU Affinity**



With SMP, the task scheduling algorithm runs on the CPU executing the kernel function that resulted in the task state change. A task state change often requires task switching on another CPU. In that case, an inter-processor interrupt (IPI) is sent to the other CPU to perform the task switching; in some cases this can trigger additional scheduling IPI requests for yet another CPU. The task state change, scheduling, and task context switches are executed with the protection of a kernel lock.

For more on scheduling, see *6.5 Task Scheduling*, p.85 and *9.18 POSIX and VxWorks Scheduling*, p.208.

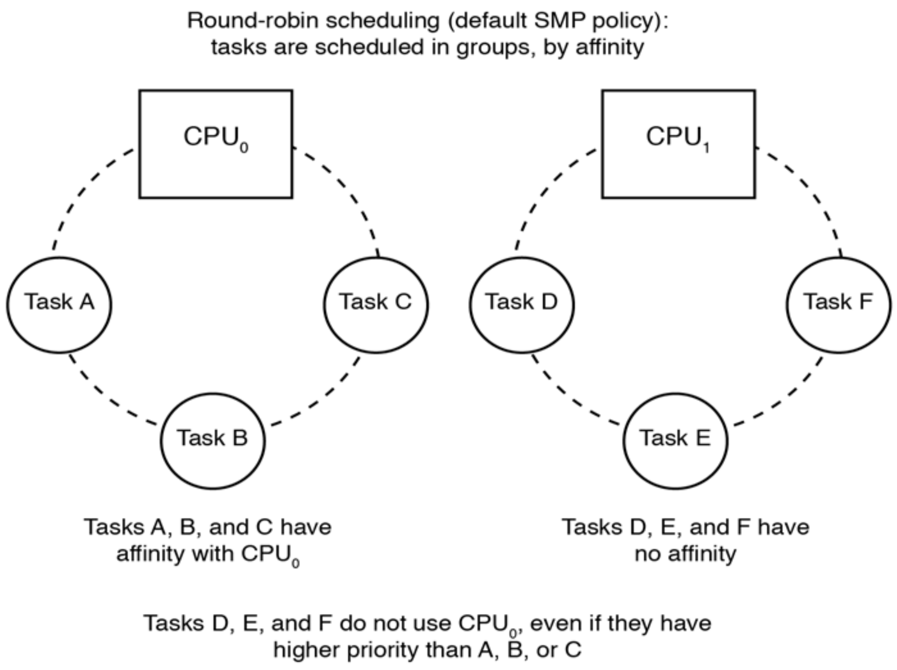
Default SMP Scheduler Policy

The default SMP scheduler policy has the following characteristics:

- Enabled with the **INCLUDE_SMP_SCHED_DEFAULT_POLICY** component
- Minimizes scheduling overhead (for example, context save and restore, cold cache, TLB)
- Makes some trade-offs to achieve more deterministic execution time and throughput, even when the enabled CPU count is high

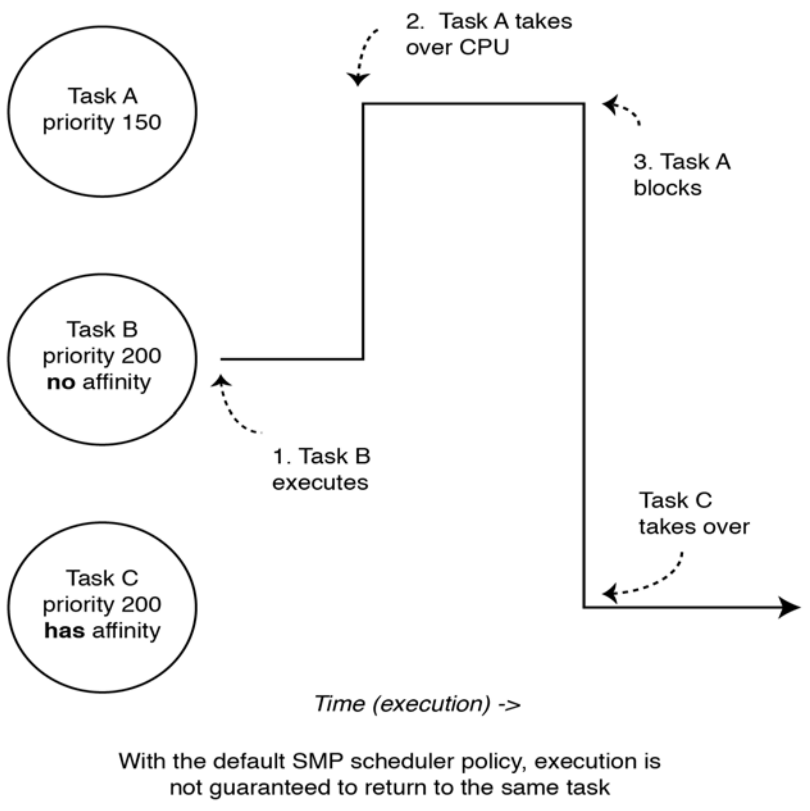
Default SMP scheduling does not ensure deterministic FIFO order for equal-priority tasks, in the following two ways. First, round-robin scheduling of equal-priority tasks occurs among tasks with CPU affinity, or without CPU affinity, but not between these two types of tasks. This means that a higher-priority task may not always be allocated to a CPU, as shown in Figure 18-5:

Figure 18-5 **SMP Default Scheduling Policy, Round-Robin and CPU Affinity**


Round-robin scheduling (default SMP policy):
tasks are scheduled in groups, by affinity
CPU$_0$
CPU$_1$
Task A
Task B
Task C
Task D
Task E
Task F
Tasks A, B, and C have affinity with CPU$_0$
Tasks D, E, and F have no affinity
Tasks D, E, and F do not use CPU$_0$, even if they have higher priority than A, B, or C

Second, a preempted task *without* CPU affinity may lose its position in favor of a task of the same priority *with* CPU affinity once the highest-priority task blocks or exits, as shown in Figure 18-6:

Figure 18-6 **SMP Default Scheduling Policy, Task Restoration**



VxWorks SMP FEPT Scheduler

As mentioned above, the default SMP scheduler does not ensure deterministic FIFO order for equal-priority tasks. For that reason, the SMP FEPT ("FIFO Equal Priority Task") scheduler exists. The FEPT policy ensures that a task added first to the ready queue will run before an equal-priority task that was added later.

Looking at Figure 18-6 again, under the SMP FEPT scheduler policy, Task B would resume execution, even though Task C has affinity.

The SMP FEPT scheduler policy generally has a higher scheduling overhead than the default policy. Moreover, the SMP FEPT policy doesn't scale as well for systems with many CPUs. This means the more CPUs, the more scheduling overhead increases compared to the default scheduler.

The FEPT scheduler is enabled by configuring VxWorks with the **INCLUDE_SMP_SCHED_FEPT_POLICY** component.

SMP SMT Scheduler Policy

The SMP SMT policy is the same as the default SMP policy, but additionally supports load balancing to optimize for processors with simultaneous multi-threading (SMT) support (for example, hyper-thread technology).

The SMP SMT scheduler policy is enabled by configuring VxWorks with the **INCLUDE_SMP_SCHED_SMT_POLICY** component.

For more information about the SMP SMT policy, see *18.16 SMP Scheduler Policy for SMT—Simultaneous Multi-Threading*, p.446.

RTP Time Partition Scheduling

The Safety Profile provides an additional scheduler that allows for scheduling RTPs in time windows; see *15.11 RTP Time Partition Scheduling*, p.388.)

18.13 CPU Information and Management

VxWorks SMP provides several functions and macros for getting and manipulating information about CPUs, as well as for managing their operation.

CPU Information and Management Functions

The **kernelLib** and **vxCpuLib** libraries provide functions for getting information about, and for managing, CPUs. Some of the key functions are described in Table 18-11 and Table 18-12.

Table 18-11 **kernelLib CPU Functions**

Function	Description
kernelIsCpuIdle( )	Returns **TRUE** if the specified CPU is idle.
kernelIsSystemIdle( )	Returns **TRUE** if all enabled CPUs are idle
kernelCpuEnable( )	Enables the CPU with the specified index.

The **kernelCpuEnable( )** function allows you to enable a specific CPU. Once a CPU is enabled, it starts dispatching tasks as directed by the scheduler. All CPUs are enabled by default, but with the **ENABLE_ALL_CPUS** parameter set to **FALSE**, VxWorks SMP is booted with just its logical CPU 0 enabled (for more information see *ENABLE_ALL_CPUS*, p.418). Then, **kernelCpuEnable( )** can be used to selectively enable individual CPUs.

Table 18-12 **vxCpuLib CPU Functions**

Function	Description
vxCpuConfiguredGet( )	Returns the number of CPUs that have been statically configured into the system with the BSP.
vxCpuEnabledGet( )	Returns the set of CPUs that are enabled in the system.
vxCpuIndexGet( )	Returns the logical CPU index for the CPU on which the function is executed.
vxCpuPhysIndexGet( )	Returns the physical CPU index for the CPU on which the function is executed.

The **vxCpuConfiguredGet( )** function returns the *number* of CPUs configured into a VxWorks SMP system with the BSP, which may not be the same as the number of CPUs available in the hardware platform.

The **vxCpuEnabledGet( )** function returns the *set* of CPUs enabled (running) on the system, of which the total may be different from the number of CPUs configured into the system with the BSP, or available in the hardware platform. As noted above, all CPUs are enabled by default, but the **ENABLE_ALL_CPUS** configuration parameter can be set so that VxWorks SMP boots with just logical CPU 0 enabled (for more information see *ENABLE_ALL_CPUS*, p. 418). Then the **kernelCpuEnable( )** function can be used to selectively enable individual CPUs.

The return type used by **vxCpuEnabledGet( )** to identify a CPU set is **cpuset_t**.

The **CPUSET_ZERO( )** macro must be used to clear the **cpuset_t** variable before the **vxCpuEnabledGet( )** call is made. For information about the **cpuset_t** variable type, and the macros used to manipulate **cpuset_t** variables, see *CPU Set Variables and Macros*, p. 442.

The function **vxCpuIndexGet( )** returns the logical index of the calling task's CPU. The index of a CPU is a number between 0 and *N*-1, where *N* is the number of CPUs configured into the SMP system. Note, however, that tasks can migrate from one CPU to another (by default), so there is no guarantee that the index returned by **vxCpuIndexGet( )** identifies the CPU on which the task is running after the return—unless the calling task is prevented from migrating to another CPU with **taskCpuLock( )** or **intCpuLock( )**.

CPU Set Variables and Macros

VxWorks SMP provides a *CPU set* variable type, and CPU set macros for manipulating variables defined by that type. The variable and macros must be used in conjunction with various functions—such as **taskCpuAffinitySet( )**—for getting information about CPUs and managing their use.

The **cpuset_t** variable type is used for identifying the CPUs that have been configured into a VxWorks SMP system with the target BSP, which may be a subset of the CPUs in the hardware platform.

Each element in a **cpuset_t** variable corresponds to a *logical* CPU index, with the first bit representing logical CPU 0. The first bit corresponds to index 0, the second to 1, the third to 2, and so on (regardless of the physical location of the CPUs in the hardware).

As an example, for an eight CPU hardware system, for which the BSP configures four CPUs for VxWorks SMP, the **CPUSET_ZERO( )** macro would clear all the elements in a **cpuset_t** variable, and then a call to **vxCpuIndexGet( )** would set the first four.

CPU set macros *must* be used to set and unset CPU indices (change the elements of **cpuset_t** variables). These macros are described in *Table 18-13CPU Set Macros*, p. 443. In order to use these macros, include the **cpuset.h** header file.

CAUTION: Do not manipulate **cpuset_t** type variables directly. Use CPU set macros.

Table 18-13 **CPU Set Macros**

Macro	Description
CPUSET_SET( )	Sets a specific CPU index (one specific **cpuset_t** variable element).
CPUSET_SETALL( )	Sets CPU indices (all **cpuset_t** variable elements) for all CPUs that are configured into the system.
CPUSET_SETALL_BUT_SELF( )	Sets indices (**cpuset_t** variable elements) for all CPUs that are configured into the system, except for the index of the CPU on which the macro is called.
CPUSET_CLR( )	Clears a specific CPU index (one specific **cpuset_t** variable element).
CPUSET_ZERO( )	Clears all CPU indices (all **cpuset_t** variable elements).
CPUSET_ISSET( )	Returns **TRUE** if the specified index (**cpuset_t** variable element) is set in the **cpuset_t** variable.
CPUSET_ISZERO( )	Returns **TRUE** if the no indices (**cpuset_t** variable elements) are set in the **cpuset_t** variable.
CPUSET_ATOMICSET( )	Atomically sets a specific CPU index (one specific **cpuset_t** variable element).
CPUSET_ATOMICCLR( )	Atomically clears a specific CPU index (one specific **cpuset_t** variable element).

For an example of how CPU set macros are used, see *Task CPU Affinity*, p.433. For more information about the macros, see the entry for **cpuset** in the VxWorks API references.

18.14 SMP Code Debugging

Debugging and system monitoring tools such as System Viewer, and the Workbench debugger provide support for debugging SMP code.

For information about debugging problems related to spinlock use, see *Debug Versions of Spinlocks*, p.426.

For information about target tool behavior with VxWorks SMP, see the *VxWorks 7 Kernel Shell User's Guide*.

As appropriate, debugging facilities provide CPU-specific information. For example, **checkStack( )** displays the interrupt stack of all CPUs. The output from the kernel shell looks like the following:

```
-> checkStack
NAME         ENTRY        TID        SIZE   CUR  HIGH  MARGIN
------------ ------------ ---------- ----- ----- ----- ------
tJobTask     0x601f3bb0   0x603ec010 24576   208   996  23580
(Exception Stack)                    12168     0   908  11260
tExcTask     0x601f35e0   0x603a4e40 24576   256   340  24236
(Exception Stack)                    12168     0   360  11808
tLogTask     logTask      0x603ec3e0 24576   320   440  24136
(Exception Stack)                    12168     0   360  11808
tShell0      shellTask    0x6058d2a0 81920   944  9976  71944
(Exception Stack)                    11328     0   908  10420
ipcom_tickd  0x600d4110   0x605919f8 24576   240   628  23948
(Exception Stack)                    12168     0   360  11808
tVxdbgTask   0x601900b0   0x60425848 24576   192   916  23660
(Exception Stack)                    11416     0   360  11056
tTcfEvents   0x602511e0   0x60388560 40960   352 10184  30776
(Exception Stack)                    11416     0   892  10524
tTcf         0x601a98e0   0x6058d788 49152  1808  2664  46488
(Exception Stack)                    11416     0   892  10524
tTcf         0x601a98e0   0x60d4d660 49152  1808  2196  46956
(Exception Stack)                    11416     0   360  11056
tTcf         0x601a98e0   0x605a4160 49152  1760  2148  47004
(Exception Stack)                    11416     0   360  11056
tAioIoTask1  aioIoTask    0x60404338 40960   224   308  40652
(Exception Stack)                    12168     0   360  11808
tAioIoTask0  aioIoTask    0x60408010 40960   224   308  40652
(Exception Stack)                    12168     0   360  11808
tNet0        ipcomNetTask 0x60489b30 24576   224  3900  20676
(Exception Stack)                    12168     0  1224  10944
ipcom_syslog 0x600d2dd0   0x60c6dac0 24576   432   836  23740
(Exception Stack)                    12168     0   908  11260
tNetConf     0x6010d980   0x603fdac8 24576   672  1924  22652
(Exception Stack)                    12168     0   360  11808
tAnalysisAge cafe_event_t 0x60374600  8192   304  1732   6460
(Exception Stack)                    11416     0   360  11056
tAioWait     aioWaitTask  0x60404010 40960   368  1012  39948
(Exception Stack)                    12168     0   360  11808
INTERRUPT                            57344     0  2700  54644
value = 63 = 0x3f = '?'
```

And **spy( )** reports the number of ticks spent in kernel, interrupt, idle, and task code for each CPU. The output looks like the following:

```
-> spy
value = 39887680 = 0x260a340 = '@'
->
NAME         ENTRY        TID        PRI   total % (ticks)  delta % (ticks)
------------ ------------ ----------  ---   ---------------  ---------------
tJobTask     0x601f3bb0   0x603ec010    0      0% (       0)    0% (       0)
tExcTask     0x601f35e0   0x603a4e40    0      0% (       0)    0% (       0)
tLogTask     logTask      0x603ec3e0    0      0% (       0)    0% (       0)
tShell0      shellTask    0x6058d2a0    1      0% (       0)    0% (       0)
tSpyTask     spyComTask   0x60594850    5      0% (       4)    0% (       0)
ipcom_tickd  0x600d4110   0x605919f8   20      0% (       0)    0% (       0)
tVxdbgTask   0x601900b0   0x60425848   25      0% (       0)    0% (       0)
tTcfEvents   0x602511e0   0x60388560   49      0% (       0)    0% (       0)
tTcf         0x601a98e0   0x6058d788   49      0% (       0)    0% (       0)
tTcf         0x601a98e0   0x60d4d660   49      0% (       0)    0% (       0)
tTcf         0x601a98e0   0x605a4160   49      0% (       0)    0% (       0)
tAioIoTask1  aioIoTask    0x60404338   50      0% (       0)    0% (       0)
tAioIoTask0  aioIoTask    0x60408010   50      0% (       0)    0% (       0)
tNet0        ipcomNetTask 0x60489b30   50      0% (       0)    0% (       0)
ipcom_syslog 0x600d2dd0   0x60c6dac0   50      0% (       0)    0% (       0)
tNetConf     0x6010d980   0x603fdac8   50      0% (       0)    0% (       0)
tAnalysisAge cafe_event_t 0x60374600   50      0% (       0)    0% (       0)
tAioWait     aioWaitTask  0x60404010   51      0% (       0)    0% (       0)
KERNEL                                         0% (       0)    0% (       0)
```

```
INTERRUPT                                       0% (       0)    0% (       0)
IDLE                                           99% (    2000)  100% (     500)
TOTAL                                          99% (    2004)  100% (     500)
```

Note that while **timexLib** can avoid precision errors by auto-calibrating itself and doing several calls of the functions being monitored, it suffers from the lack of scheduling management during the calls. The tasks can move between CPUs while the measurements take place. Depending on how often this occurs, this is likely to have an impact the precision of the measurement.

18.15 SMP Performance Optimization

The purpose of SMP systems is to increase performance. Simply making code *SMP-ready* does not necessarily exploit the performance potential available with multiple CPUs. Additional work is necessary to make code *SMP-optimized*.

For definitions of these terms, see *Terminology*, p.412.

The performance improvement of an SMP algorithm is almost completely dependent on the amount of parallelism in the algorithm and the quality of the multi-threaded implementation. Some algorithms are highly parallel in nature, and take good advantage of multiple CPUs. A good example is an image compressor, which can compress separate bands of data independently on separate threads of execution. Since contention is low, the utilization of the CPU can be very high, resulting in good SMP performance.

With a poorly-designed SMP algorithm, on the other hand, the cost of synchronizing two threads of execution can completely negate the benefits of using more than one CPU. Similarly, if you have a data-dependent algorithm and both CPUs are competing for the same data, the system bus can be swamped with competing bus transactions, slowing the throughput to the point that the CPUs are data-starved, and overall throughput suffers.

In the worst case, SMP will actually slow down an algorithm, resulting in worse performance than on a uniprocessor system. In the best case, taking advantage of the fact that there is twice as much L1 cache in a dual-processor system, might allow algorithms to run twice as fast, simply because the working set of the algorithm fits better in the twice-as-large cache. These types of algorithms are, however, fairly rare.

Threading

Threading involves turning a single-thread application into a multi-threaded one by replicating tasks. A typical example involves a worker task that fetches work from a queue that is being filled by another task or an ISR. Assuming the bottleneck in the application is the worker task, performance can be increased by replicating the worker task. Threading is not a new concept—it was introduced when multitasking operating systems were created. However, in a uniprocessor system threading only increases throughput of the application when its threads are subject to wait periods. That is, when one thread waits for a resource, the scheduler can dispatch another thread, and so on. In cases where the bottleneck is the CPU itself, threading cannot help performance. For example, compute intensive applications typically do not benefit from threading on a uniprocessor system. However, this is not the case on a SMP system. Because of the presence of additional CPUs, threading increases performance particularly when the bottleneck is the CPU.

Using Spinlocks

Spinlocks affect interrupt and task-preemption latency and should therefore be used sparingly and only for very short periods of time. For more information, see *18.5 Spinlocks for Mutual Exclusion and Synchronization*, p.422.

Using Floating Point and Other Coprocessors

For reasons of efficiency, coprocessor task creation options (such as **VP_FP_TASK**) should be used carefully—that is, only with tasks that will actually make use of coprocessors. When a task is created with a coprocessor option, the state of the coprocessor is saved and restored with each context switch, which is unnecessary overhead if the coprocessor is never used. VxWorks SMP does not support the VxWorks UP option of *lazy state save-and-restore*, because tasks are not guaranteed to resume execution on the same CPU on which they were last scheduled to run.

Using vmBaseLib

The **vmBaseLib** library is the VxWorks MMU management library that allows kernel applications and drivers to manage the MMU. An important task of an SMP operating system is to ensure the coherency of the translation look aside buffers (TLBs) of the MMU contained in each CPU. Some CPUs, like the MPC8641D, have hardware that ensures TLBs are always coherent. Other CPUs, such those in the MIPS architecture family, do not have this capability. In these cases the operating system is responsible for propagating MMU events that affect TLB coherency to all CPUs in the system.

While not all events require propagation—it is generally limited to events that modify an existing page mapping such as with **vmStateSet( )—the** propagation that must be performed has a negative impact on some VxWorks SMP **vmBaseLib** functions. To reduce the negative impact on your system's performance, minimize the number of calls to **vmStateSet( )**, and so on. For example, if a region with special settings is needed from time to time during system operation, it is better to set it up once during startup, and then reuse it as needed, rather than creating and destroying a region for each use.

Interrupt and Task CPU Affinity

For some applications and systems, assigning specific interrupts or specific tasks to designated CPUs can provide performance advantages. For more information, see *Interrupt CPU Affinity*, p.434 and *Task CPU Affinity*, p.433.

18.16 SMP Scheduler Policy for SMT—Simultaneous Multi-Threading

VxWorks SMP provides an alternate, load-balancing scheduler policy to improve the performance of simultaneous multi-threading (SMT) on superscalar processors.

Scheduling is in all other respects the same as with the default SMP scheduler. The SMT scheduler policy is provided with the **INCLUDE_SMP_SCHED_SMT_POLICY** component. For more information about configuration, see *SMP Scheduler Policies*, p.419.

About Simultaneous Multi-Threading

Simultaneous multi-threading (SMT) is a technique for improving the overall performance of superscalar processors with *hardware multithreading*. With SMT processor design, specific processor units (such as the instruction dispatch unit and register set) are replicated for each SMT thread. The remaining units (such as the cache, TLB, the floating point unit, and other execution units) are shared by two or more SMT (hardware) threads. The replication of units allows a single core to simultaneously dispatch instructions from more than one SMT thread contexts (producing thread parallelism). In effect then, each core can function as two or more complete CPU cores (depending on the architecture). SMT is transparent to the operating system and applications.

The thread-level parallelism provided by SMT can produce notable performance gains at the cost of a relatively modest increase in silicon footprint and power consumption. Vendor-specific examples of SMT include the Hyper-Threading Technology (Intel) and nxCPU (originally developed by Netlogic).

VxWorks SMT Scheduler Policy

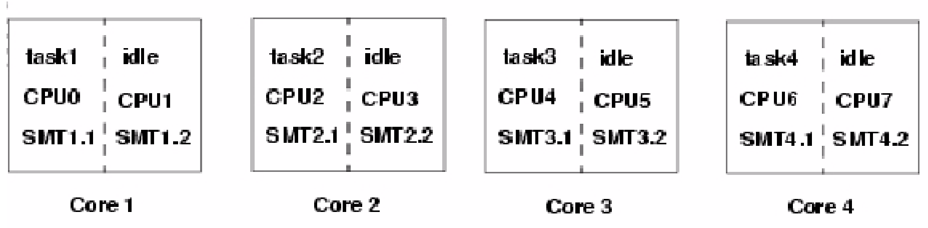
SMT does introduce an *asymmetric* aspect to SMP systems—when multiple tasks are scheduled on SMT threads sharing the same physical core, their performance is slightly lower than when they are run on different physical cores. This is because SMT threads on the same physical core share some of the same execution units, cache, and so on.

The default VxWorks SMP scheduler does not differentiate between cores with SMT threads and cores without SMT threads, and therefore does not address SMT asymmetry.

VxWorks does, however, provide an SMT scheduler policy that can be used in place of the default scheduler policy. The SMT scheduler policy distributes the work load evenly across physical cores, minimizing the impact of sharing resources.

In the example illustrated with Figure 18-7, the system has four physical cores—each with two SMT threads—and four ready tasks. The SMT scheduler policy assigns each task to a different core, leaving the second SMT thread of each core idle, making the most efficient use of hardware resources.

Figure 18-7 **SMT Scheduler Policy**



If a system is fully loaded with ready tasks, no load balancing is possible—for example, if there are eight or more ready tasks and a total of eight SMT threads.

If a system contains only one core (with two or more SMT threads), the VxWorks SMT scheduler policy cannot do any meaningful load balancing and should not be used. Resources are necessarily shared whenever there is more than one ready task.

SMT Scheduler Policy and Task Priorities

When a task becomes ready, the scheduler assigns it to a SMT thread belonging to the physical core with the lightest load. The scheduler initiates task migration from one core to another if the difference between the loads of the two cores is at least two.

The load balancing provided by the SMT scheduler policy does not take task priority into consideration—high priority tasks are treated in the same way as low priority tasks.This means that high priority tasks may not end up in the most optimal position. For example, in a two core system, a high priority task and a low priority task might be assigned to one core (and thus necessarily share resources), while a sole low priority task is assigned to the second core.

However, VxWorks SMP and CPU affinity and CPU reservation can be used to ensure that high priority tasks do not share physical cores. Reserved CPUs are excluded from SMT load balancing, so using CPU reservation with task CPU affinity allows you to override automatic load balancing for a subset of CPUs. In this way, for example, reserving all CPUs corresponding to the SMT threads of a physical core for one or more high priority tasks ensures that they will never share resources with lower priority tasks, and never suffer the associated performance degradation. For information about CPU affinity and reservation, see *18.10 CPU Affinity*, p.432 and *18.11 CPU Reservation for CPU-Affinity Tasks*, p.435.

18.17 Sample Programs

Sample programs are provided for VxWorks SMP, which demonstrates VxWorks SMP features and performance.

Applications Illustrating I/O and System-call Intensive Activity

Application	Description
philDemo	Dijkstra's *Dining Philosophers Problem*.
smpLockDemo	Demonstrates VxWorks SMP's synchronization mechanism for sharing data across multiple CPUs.
primesDemo	Prime number computation.
rawPerf	Calculation of *pi* using floating-point arithmetic.
smtPerfDemo	Performs discrete Fourier transform using integer arithmetic to show the effectiveness of the SMT scheduling policy.

The demo applications can be linked to a VxWorks SMP image by configuring VxWorks with the **INCLUDE_SMP_DEMO** component. For more information about the demos, refer to their entries in the *VxWorks Kernel API Reference*.

18.18 Code Migration for VxWorks SMP

The key issue to consider in migrating code to an SMP system is that symmetric multiprocessing allows for concurrent execution of tasks with other tasks, of tasks with ISRs, and of ISRs with other ISRs.

Concurrent execution in SMP requires the use of different facilities for mutual exclusion and synchronization, it precludes some functions that are available for a uniprocessor system, and it makes the practice of relying on implicit synchronization of tasks dangerous (if not disastrous). For example, the concurrent execution of several of the highest priority tasks in a system can uncover unprotected race conditions that were hidden in a uniprocessor system.

In addition, multiple CPUs introduce complexities with regard to objects that are *global* in a uniprocessor system, but must be *CPU-specific* in an SMP system.

Migrating code from VxWorks UP to VxWorks SMP necessarily involves several steps between uniprocessor code and hardware to SMP code and hardware.

The migration process also involves using different multitasking facilities, different BSP support, and so on. Some parts of migration activity involve replacing a uniprocessor technology with an SMP one—such as replacing **taskLock( )** with **spinLockTaskTake( )**—while others involve changing the use of features that have different behaviors in the VxWorks UP and VxWorks SMP (for example, some **vmBaseLib** functions).

This section provides an overview of the migration process, a summary of the operating system facilities that need to be taken into account in migration, and more detailed information about individual migration issues. It does not provide a completely self-contained discussion of migration to SMP. It is necessarily a supplement to the preceding material in this chapter, which provides information about the core features of VxWorks SMP. Incorporation of these features naturally forms the basis for migrating code from VxWorks UP to VxWorks SMP. The material following the discussion of general issues—*Code Migration Path*, p. 449 and *Overview of Migration Issues* , p. 450—therefore covers some of the less tidy aspects of migration.

CAUTION: Code built for VxWorks is binary compatible only if it is based on the same VSB configuration (with the same set of layers and versions). In addition, kernel C++ code must be built with the same compiler as the VxWorks image.

Code Migration Path

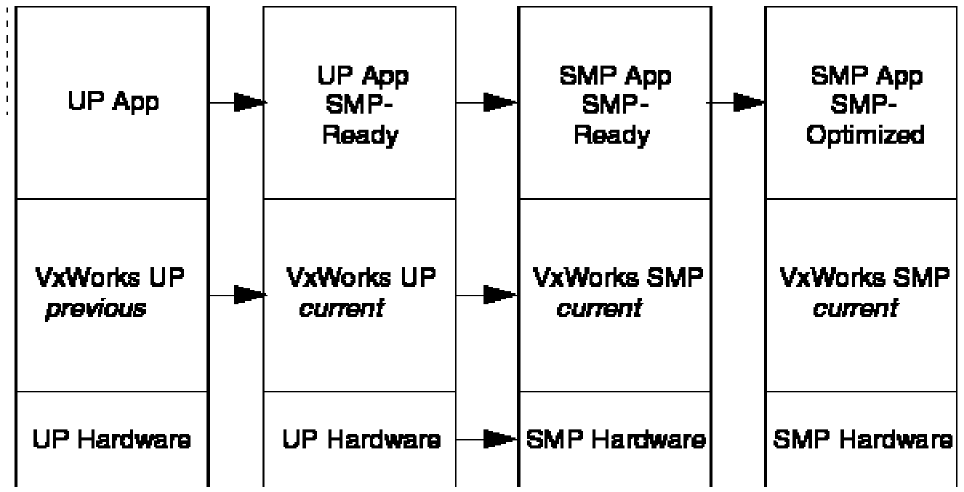
Wind River recommends that you approach migrating code designed for an earlier version of VxWorks UP to the current version of VxWorks SMP with the following steps:

1. Migrate Uniprocessor Code from Previous to Current Version of VxWorks

2. Migrate Code from Current VxWorks UP to VxWorks SMP
3. Optimize Code for SMP Performance

Figure 18-8 illustrates the recommended application migration path.

Figure 18-8 **VxWorks SMP Migration**



Migrating code written for an earlier VxWorks UP release to a current VxWorks UP release should include eliminating or replacing any functions and coding practices that are unsupported or incompatible with VxWorks SMP. For an overview of what must be replaced or changed, see *Overview of Migration Issues* , p.450. Note that this step might include migrating from uniprocessor hardware to a single processor on SMP hardware; that is, the hardware used for the end-point of the first and second migration steps shown in Figure 18-8 may therefore be the same.

Migrating the code that you have adapted to a current VxWorks UP release (and made compatible with VxWorks SMP) to a current VxWorks SMP release involves correcting any concurrent execution bugs (such as contention issues and deadlocks) that appear in an SMP environment. This step includes migrating to multiprocessor use of SMP hardware.

Optimizing the code on VxWorks SMP allows it to make the fullest use of symmetric multiprocessing. For more information about this topic, see *18.15 SMP Performance Optimization*, p.445.For definitions SMP-ready and SMP-optimized see *Terminology*, p.412.

Overview of Migration Issues

Table 18-14 provides an overview of the uniprocessor features or programming practices that are incompatible with (or unsupported by) VxWorks SMP, the appropriate SMP alternatives, and references to these topics. All code designed for VxWorks UP should be examined carefully with regard to these issues as part of the migration process.

Table 18-14 **VxWorks SMP Migration Issues**

Incompatible Uniprocessor Features and Practices	SMP Features and Practices	Discussion
Coding practices relying on implicit synchronization.	Use of explicit synchronization facilities, such as semaphores and spinlocks.	*Implicit Synchronization of Tasks*, p.452
Various **cacheLib** functions.	Revise use of functions.	*cacheLib Restrictions*, p.453
Various **vmBaseLib** functions.	Restrict use of functions.	*vmBaseLib Restrictions*, p.454
taskLock( ), **intLock( )**	**spinLockLib**, **taskCpuLock( )**, **intCpuLock( )**, atomic operators	*Synchronization and Mutual Exclusion Facilities*, p.452 and *Task Locking: taskLock( ) and taskUnlock( )*, p.455
taskRtpLock( ), **taskRtpUnlock( )** (Functions deprecated as of VxWorks 6.7 and not provided with 64-bit VxWorks)	semaphores, atomic operators	*Synchronization and Mutual Exclusion Facilities*, p.452 and *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p.455
task variables, **taskVarLib** functions	**__thread** storage class	*Task Variable Management: taskVarLib* , p.456
tlsLib functions	**__thread** storage class	*Task Local Storage: tlsLib*, p.456
Accessing global variables that are *CPU-specific* variables or inaccessible in SMP.	Replace with *CPU-specific* variable functions and practices.	*SMP CPU-Specific Variables and Uniprocessor Global Variables*, p.456
Memory-access attributes unsuited for SMP memory coherency.	Review calls that directly manipulate coherency protocols and caching modes. Adhere to restrictions as documented in architecture supplements.	*Memory-Access Attributes*, p.458
Drivers that are not VxBus-compliant.	VxBus-compliant drivers.	*Drivers and BSPs*, p.458
Uniprocessor BSP.	SMP BSP.	*Drivers and BSPs*, p.458
Uniprocessor boot loader.	Boot loader that supports VxWorks SMP.	

Also note that the custom scheduler framework is not supported for VxWorks SMP.

RTP Applications and SMP

As in VxWorks UP systems, RTP (user mode) applications have a more limited set of mutual exclusion and synchronization mechanisms available to them than kernel code or kernel applications. In VxWorks SMP, they can make use of semaphores and atomic operations, but not spinlocks, memory barriers, or CPU-specific mutual exclusion mechanisms. In addition, the **semExchange( )** function provides for an atomic give and exchange of semaphores.

Implicit Synchronization of Tasks

VxWorks is a multitasking operating system, and VxWorks and its applications are re-entrant; therefore migrating to a system in which tasks run concurrently is not normally a problem as long as tasks are explicitly synchronized. For example, Task A giving a semaphore to Task B to allow it to run is a form of explicit synchronization. On the other hand, implicit synchronization techniques—such as those that rely on task priority—cannot be relied on in VxWorks SMP. For example, if high priority Task A spawns low priority Task B, expecting that Task B will not run until Task A releases the CPU is an invalid assumption on an SMP system.

Implicit synchronization based on task priority is not easy to detect. Careful review of all code that causes a task to become ready to run would be a useful approach. For example, review code that uses the following types of functions:

- Functions That Create Tasks
- Functions that Unpend a Waiting Task

The **taskSpawn( )**, **rtpSpawn( )**, and other functions create a new task. In an SMP system, the new task may have already started running on another CPU by the time the procedure call returns—regardless of the relative priority of the new task compared to the creating task. If the creator and the created tasks interact using semaphores, message queues or other objects, these objects *must* be created or initialized before creating the new task.

The **semGive( )**, **msgQSend( )**, **eventSend( )** and other functions can unpend a waiting task, which may begin running before the procedure call returns—even though the waiting task has a lower priority than the calling task.

For example, in a VxWorks UP system, a task can protect a critical section of code by using **intLock( )**, which prevents all interrupts from being processed and thereby prevents ISRs from entering the critical section. The ISR does not use explicit mutual exclusion when accessing the critical section because the task cannot be running when the ISR is running on a uniprocessor system. (This is likely to be a common occurrence in drivers where a portion of a driver runs in an ISR and queues up work for a task to perform.) The assumption that tasks do not run when ISRs do is simply not true in an SMP system. Therefore ISRs must use explicit mutual exclusion in cases such as the one described above. The preferred mechanism is the ISR-callable spinlock as described in *18.5 Spinlocks for Mutual Exclusion and Synchronization*, p.422.

Synchronization and Mutual Exclusion Facilities

Because of concurrent execution on an SMP system, there are necessarily differences in facilities available for explicit synchronization and mutual exclusion for VxWorks UP and VxWorks SMP.

Semaphores are appropriate for both environments, but uniprocessor interrupt and task locking mechanisms are not appropriate and not available for SMP—spinlocks and other mechanisms should be used instead.

In VxWorks SMP it is, for example, possible for a task to be running at the same time that an ISR is executing. This is not possible in VxWorks UP, and therefore requires changes to the way mutual exclusion between a task and an ISR is done. A common synchronization method between an ISR and a task in VxWorks is the binary semaphore. This mechanism works equally well in VxWorks SMP, and therefore code that uses binary semaphores in this manner need not be modified for VxWorks SMP—provided the ISR is running with interrupts enabled when it calls **semGive( )**. This is also true of other messaging and synchronization functions, such as message queues and VxWorks events. Note, however, that when an ISR wakes up a task (by giving a binary semaphore, sending a VxWorks event, sending a message to a message queue, etc.), the awakened task may start running *immediately* on another CPU.

For more information about uniprocessor synchronization mechanisms and the SMP alternatives, see *Unsupported Uniprocessor Functions and SMP Alternatives* , p. 454.

Note that VxWorks provides scalable and inline variants of the give and take functions for binary and mutex semaphores. These variants provide alternatives to the standard functions that are particularly useful for SMP systems (for more information in this regard, see *Scalable and Inline Semaphore Take and Give Kernel Functions*, p. 119).

VxWorks SMP Variants of Uniprocessor Functions

While the functions provided in VxWorks UP and VxWorks SMP are largely the same, there are a few that have different behaviors in the VxWorks SMP due to the requirements of multiprocessor systems, and their use has restrictions.

cacheLib Restrictions

The VxWorks UP cache functions are designed around a uniprocessor system. Enabling and disabling the caches, invalidating, flushing or clearing elements of the cache all have a CPU-specific nature, as they refer to the local CPU's cache. In an SMP system, this CPU-specific nature is less meaningful. The systems that are supported by VxWorks SMP all provide hardware cache coherency, both between the individual CPUs in the SMP system and between the memory subsystem and the device address space. Given these characteristics, the cache restrictions and behavior modifications described below apply to VxWorks SMP.

cacheEnable( ) and cacheDisable( )

The only way for the hardware cache coherency to be effective is to have the caches turned on at all times. VxWorks SMP therefore turns on the caches of each CPU as it is enabled, and never allows them to be disabled. Calling **cacheEnable( )** in VxWorks SMP always returns **OK**. Calling **cacheDisable( )** in VxWorks SMP always returns **ERROR**, with errno set to **S_cacheLib_FUNCTION_UNSUPPORTED**.

cacheClear( ), cacheFlush( ), and cacheInvalidate( )

Because of the hardware cache coherency of SMP-capable platforms, these functions are not necessary. If these functions are called in VxWorks SMP, they perform no function (are NOOPs) and simply return **OK**.

cacheLock( ) and cacheUnlock( )

These functions are not supported in VxWorks SMP. If they are called, they return **ERROR** and set errno to **S_cacheLib_FUNCTION_UNSUPPORTED**.

cacheTextUpdate( )

On some hardware architectures, cache coherency is not maintained between the instruction caches and the data caches between the various CPUs. To address this lack of coherency, **cacheTextUpdate( )** is still required for VxWorks SMP whenever its use is called for on VxWorks UP.

vmBaseLib Restrictions

VxWorks SMP does not provide APIs for changing memory page attributes. On an SMP system it is essential that the RAM regions that are shared between the CPUs never be allowed to get out of coherency with one another. If a single page in system RAM were to have its attributes changed so that it no longer correctly participates in the hardware coherency protocol, any operating system use of that page (for spinlocks, shared data structures, and so on) would be at risk of unpredictable behavior. This unpredictable behavior might even occur long after the offending change in the page attributes has occurred. This type of problem would be extremely difficult to debug, because of the underlying assumption that the hardware coherency in SMP simply works.

vmBaseStateSet( ) and vmStateSet( )

These functions are called to modify the attributes of a single page of virtual memory. In an SMP system, the caching attributes of a page cannot be modified. Attempting to do so causes these functions to return **ERROR** with errno set to **S_vmLib_BAD_STATE_PARAM**.

Unsupported Uniprocessor Functions and SMP Alternatives

Some of the functions available in VxWorks UP are not supported in VxWorks SMP because their functionality is at odds with truly concurrent execution of tasks and ISRs, or because they would degrade performance to an unacceptable extent. SMP alternatives provide comparable functionality that is designed for symmetric multiprocessing.

Interrupt Locking: intLock( ) and intUnlock( )

In VxWorks UP, the **intLock( )** function is used by a task or ISR to prevent VxWorks from processing interrupts. The typical use of this function is to guarantee mutually exclusive access to a critical section of code between tasks, between tasks and ISRs, or between ISRs (as with *nested ISRs*—when ISR can be preempted by an ISR of higher priority).

This mechanism would be inappropriate for a multiprocessor system, and VxWorks SMP provides the following alternatives for interrupt locking:

- If interrupt locking is used to make a simple pseudo-atomic operation on a piece of memory, atomic operations may be a suitable alternative.
- If interrupt locking is used as a mutual exclusion mechanism between tasks only, semaphores or task-only spinlocks are a suitable replacements. Spinlock acquisition and release operations are faster than semaphore operations, so they would be suitable to protect a short critical section that needs to be fast. Semaphores are suitable for longer critical sections.
- If interrupt locking is used as a mutual exclusion mechanism between tasks and ISRs, or between ISRs, ISR-callable spinlocks are a suitable replacement.
- If interrupt locking is used as a mutual exclusion mechanism between tasks only, **taskCpuLock( )** can be used instead as long as all tasks taking part in the mutual exclusion scenario have the same CPU affinity. This alternative should not be used in custom extensions to the operating system other than as a temporary measure when migrating code the from VxWorks UP to VxWorks SMP.

- If interrupt locking is used as a mutual exclusion mechanism between tasks, between tasks and ISRs, or between ISRs, then **intCpuLock( )** can be used as long as all tasks and ISRs taking part in the mutual exclusion scenario have the same CPU affinity. This alternative should not to be used in custom extensions to the operating system other than as a temporary measure when migrating the code from VxWorks UP to VxWorks SMP.

Note that for VxWorks SMP, ISR-callable spinlocks are implemented with the same behavior as the interrupt locking functions **intLock( )** and **intUnlock( )**.

For information about SMP mutual exclusion facilities, see *18.5 Spinlocks for Mutual Exclusion and Synchronization*, p. 422, *18.6 CPU-Specific Mutual Exclusion*, p. 428, and *18.8 Atomic Memory Operations*, p. 431.

Task Locking: taskLock( ) and taskUnlock( )

In VxWorks UP, task locking functions are used by a task to prevent the scheduling of any other task in the system, until it calls the corresponding unlock function. The typical use of these functions is to guarantee mutually exclusive access to a critical section of code.

With VxWorks UP, the kernel function **taskLock( )** is used to lock out all other tasks in the system by suspending task preemption (also see *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p. 455). This mechanism would be inappropriate for a multiprocessor system, and VxWorks SMP provides the following alternatives:

- Semaphores.
- Atomic operations.
- task-only spinlocks. Spinlock acquisition and release operations are faster than semaphore operations (the other alternative in this case) so they would be suitable to protect a short critical section that needs to be fast.
- The **taskCpuLock( )** functions for situations where all tasks taking part in the task-locking scenario have the same CPU affinity. This alternative should not be used in custom extensions to the operating system other than as a temporary measure when migrating the code from VxWorks UP to VxWorks SMP.

Note that for VxWorks UP, task-only spinlocks are implemented with the same behavior as the task locking functions **taskLock( )** and **taskUnlock( )**.

For information about SMP mutual exclusion facilities, see *18.5 Spinlocks for Mutual Exclusion and Synchronization*, p. 422, *18.6 CPU-Specific Mutual Exclusion*, p. 428, and *18.8 Atomic Memory Operations*, p. 431.

Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )

NOTE: The **taskRtpLock( )** and **taskRtpUnlock( )** functions are deprecated as of VxWorks 6.7 and are and not provided with 64-bit VxWorks.

The **taskRtpLock( )** function is used in RTP applications to prevent scheduling of any other tasks in the process of the calling task. As with **taskLock( )**, the **taskRtpLock( )** function is not appropriate for an SMP system (also see *Task Locking: taskLock( ) and taskUnlock( )*, p. 455).

The **taskRtpLock( )** function is provided with VxWorks SMP, but it generates a fatal error when called, and the process terminates. Semaphores or atomic operators should be used instead.

Task Variable Management: taskVarLib

The VxWorks UP task variable facility provided by **taskVarLib** is not compatible with an SMP environment, as more than one task using the same task variable location could be executing concurrently. Therefore task variables and the **taskVarAdd( )** and **taskVarDelete( )** functions are not available in VxWorks SMP. The **__thread** storage class should be used instead. For more information, see *Task-Specific Variables*, p. 107.

Task Local Storage: tlsLib

The VxWorks UP task local storage functions provided by **tlsLib** for user-mode (RTP) applications are not compatible with an SMP environment, as more than one task using the same task variable location could be executing concurrently. The **tlsLib** functions are as follows:

- **tlsKeyCreate( )**
- **tlsValueGet( )**
- **tlsValueSet( )**
- **tlsValueOfTaskGet( )**
- **tlsValueOfTaskSet( )**

The **__thread** storage class should be used instead. For more information, see *Task-Specific Variables*, p. 107.

SMP CPU-Specific Variables and Uniprocessor Global Variables

Some objects that are global in a uniprocessor system (such as errno) are CPU-specific entities in VxWorks SMP, and others are inaccessible or non-existent in VxWorks SMP.

⚠ **CAUTION:** Wind River recommends that you do not manipulate any CPU-specific or global variables directly. Using the appropriate API is recommended to prevent unpredictable behavior and to ensure compatibility with future versions of VxWorks.

SMP Per-CPU Variables

The SMP CPU-specific variables that can be accessed indirectly with appropriate function are as follows:

- **errno**
- **taskIdCurrent**
- **intCnt**
- **isrIdCurrent**

errno

From a programming perspective errno behaves like a global variable that contains the error value of the currently running task or ISR. VxWorks SMP has mechanism that allows it to manage errno as a CPU-specific variable in a transparent manner.

Wind River recommends that you use **errnoLib** functions to work with errno. However, you may access the errno variable directly from C and C++ code that includes **errno.h**. Do not access the errno variable directly from assembly code.

⚠ **CAUTION:** Do not access errno directly from assembly code. Do not access errno directly from C or C++ code that does not include **errno.h**.

taskIdCurrent

The uniprocessor **taskIdCurrent** global variable (declared in **taskLib.h**) does not exist in VxWorks, because of concurrent execution on multiple CPUs. Any uniprocessor code that reads **taskIdCurrent** should make calls to **taskIdSelf( )** instead.

intCnt

In an SMP system, specific interrupts are dedicated to a specific CPU. The **intCnt** variable is used to track the number of nested interrupts that exist on a specific CPU. Code that references this variable should be changed to use the **intCount( )** function instead.

isrIdCurrent

The **isrIdCurrent** variable is used to identify the ISR executing on the specific CPU. This global is only available if the **INCLUDE_ISR_OBJECTS** component is included in VxWorks. Code that accesses **isrIdCurrent** must be changed to use the **isrIdSelf( )** function instead.

Uniprocessor-Only Global Variables

The VxWorks UP variables that do not exist in VxWorks SMP—or that must not be accessed by user code in any way—are as follows:

- **vxIntStackBase**
- **vxIntStackEnd**
- **kernelIsIdle**
- **windPwrOffCpuState**

vxIntStackBase

The **vxIntStackBase** variable identifies base of the interrupt stack used for processing interrupts. For VxWorks SMP, each CPU has a **vxIntStackBase** to process interrupts since interrupts may be processed by multiple CPUs simultaneously. There is no function for accessing this variable and it must not be accessed by user code.

vxIntStackEnd

The **vxIntStackEnd** variable identifies the end of the interrupt stack for each CPU. There is no function for accessing this variable and it must not be accessed by user code.

kernelIsIdle

In VxWorks UP the **kernelIsIdle** variable indicates whether or not the system is idle. This variable does not exist in VxWorks SMP. *CPU Information and Management Functions*, p. 441 describes functions that can be used instead.

windPwrOffCpuState

The **windPwrOffCpuState** variable identifies power management state on the specific CPU. There is no function for accessing this variable and it must not be accessed by user code.

Memory-Access Attributes

In an SMP system memory coherency is required to ensure that each CPU sees the same memory contents. Depending on the CPU architecture, some memory access attributes may not be suitable for a system where memory coherency is required. For information in this regard, see the *VxWorks Architecture Supplement*.

Drivers and BSPs

Both drivers and BSPs developed for VxWorks SMP must adhere to the programming practices described throughout this chapter. Drivers must also conform to the VxBus driver model. BSPs, in addition to providing support for VxBus, must provide facilities different from the VxWorks UP for reboot handling, CPU enumeration, interrupt routing and assignment, and so on.

19
Custom System Calls

19.1 About VxWorks System Calls

The VxWorks system call interface provides kernel services for user-mode applications that are executed as real-time processes (RTPs). The interface can be extended by developers who wish to add custom system calls to the operating system to support the special needs of their applications.

NOTE: Before deciding to add custom system calls, you should determine whether or not other mechanisms that allow for communication between the user (RTP) environment and the kernel environment are sufficient for your needs. Public message queues, shared data regions, and **ioctl( )** can be used for this purpose.

The key elements of the VxWorks system call facility are the following:

- system call
- trap handler
- system call handler

A system call is a C-callable function that is implemented as short piece of assembly code. System calls differ only in their name and system call number—which are provided by developers— but the code itself is provided by the system call infrastructure.

When user application (RTP) code invokes a system call, it executes a trap instruction (software interrupt), which switches the execution mode of the calling task from user to supervisor (kernel) mode. The system call number is passed to the kernel to identify the system call and its associated system call handler function.

In the kernel, a trap handler copies any system call arguments from the registers and (or) user stack—as appropriate for the architecture—to the kernel stack. The trap handler then calls the system call handler. The trap handler is provided by the system call infrastructure. The system call handler function is written by the system call developer.

Each system call handler function is passed only one argument: the address of a structure whose members are the arguments of the system call. The system call handler must validate the structure and its members before using the member data.

A system call handler makes calls to kernel functions to provide services to the calling application. There are, however, restrictions on what functions may be used—primarily to avoid conflicts resulting from violation of the separation of the kernel and RTP environments.

When the system call handler function returns, the execution mode of the calling task is switched from supervisor (kernel) to user mode.

A system call handler's return value must be **ERROR** (-1) in case of an error status, but may otherwise be of any 32-bit type (return values of some 64-bit pointer and integer types are allowed on VxWorks 64-bit only). The system call handler’s return value is passed back to the RTP side and returned by the system call.

19.2 Custom System Call Development

Initially, the main tasks required for extending the VxWorks system call interface are designing the custom system call in accordance with the requirements for VxWorks, and then writing the system call handler to support that design. The system call can then be added to VxWorks either statically or dynamically.

Static addition involves using the same configuration and build mechanisms that are used for VxWorks system calls, and the custom system call is built into the VxWorks image.

Dynamic addition does not use these mechanisms, and requires writing additional kernel code that is used to register the system call and its handler at runtime. Dynamic addition allows for using downloadable kernel modules (DKMs) to add system calls to VxWorks, which can be registered on demand; or for linking the module with VxWorks and configuring the system to initiate registration at boot time.

Custom System Call Design

If your RTP applications need more functionality than provided by the VxWorks system calls, you can extend the standard system calls to create custom system calls. To do this work, you need to understand how the VxWorks layer infrastructure supports multiple versions of a feature and company-specific variants of the product layers.

The custom system call definition files reside in a dedicated layer. By being encapsulated in a layer, a set of custom system call definitions is automatically referred to by name and version number, which allows for its selective inclusion in a VSB. You can easily update custom system call definitions by creating a new version of the layer (which implies an updated copy of the layer content). This new version of the layer can be located anywhere you want.

In addition to the definition files, the custom system call layer can optionally host all of the other items required to use custom system calls in VxWorks. This way, all of the system call definition files can be co-located in the source tree. For example, the custom system call layer can contain:

- The custom system call handlers (source and header files for the kernel environment).
- The RTP-side header files defining the interfaces of those custom system calls.
- The source code for RTP applications using those custom system calls.

For more information about VxWorks layers, see the *VxWorks Configuration and Build Guide*.

Custom System Call Layer Layout and Content

The primary purpose of the custom system call layer is to host all of the elements that make possible the addition of custom system calls, which can then be used in RTPs. A secondary purpose of the custom system call layer is to generate the system call infrastructure.

The custom system calls layer is a sublayer of the core layer and is located in the *installDir*/**vxworks-7/pkgs/os/core/syscalls/custom** directory. The **syscalls** directory is at the same level as the **kernel** and **usr** directories. The custom system call layer is located in the **custom** directory. The VxWorks system call layer is located in the **vxworks** directory.

The following table lists the subdirectories and content under the **custom** directory.

Table 19-1 **Subdirectories and Content of the custom Directory**

Subdirectory	Description
kernel_h	Headers for type definitions needed by the system call handlers (optional)
kernel_src	Source of the system call handlers and makefile
pre_src	System makefile calling the system call infrastructure generation (**scgen**) tool. It is a system file; do not modify it.
syscall_defs	Custom system call definition files, **syscallUsrNum.def** and **syscallUsrApi.def**
user_h	Headers providing the prototypes of the custom system calls (optional). System calls are usually invoked from the RTP side directly through their function names. As a result, you must provide system call prototype definitions in the custom system call layer.

The vxworks directory contains the **syscall_defs** directory. It contains the VxWorks system call definition files, **syscallNum.def** and **syscallApi.def**.

The **custom** and **vxworks** directories also contain the layer definition file, **layer.vsbl**, and the top level makefile that defines the elements involved in the pre-build, build, and post-build phases.

For more information about the makefile macros for VxWorks layers, see the *VxWorks Configuration and Build Guide*.

19.3 Static Addition of System Calls

Custom system calls can be added to VxWorks statically, using the same mechanisms that are used for VxWorks system calls. Using this method, a VxWorks image supports additional system calls in a self-contained way. In contrast to the method of adding system calls dynamically, no additional code must be used to register system call handlers at runtime.

About System Call Definition Files

The process of adding custom system calls statically is based on the use of system call group definition and system call definition files (**syscallUsrNum.def** and **syscallUsrApi.def**), which you create from templates. These files define the system call names and system call function numbers, their prototypes, the system call groups to which they belong, and (optionally) the components with which they should be associated. The files are used to generate the system call apparatus that is required to work with the system call handlers. For more information, see *19.3 Static Addition of System Calls*, p.462.

Overview of Adding System Calls Statically

The basic steps required to add a new system call to VxWorks are as follows:

- *Step 1:Copy the Default Layer*, p.466.
- *Step 2:Update the VERSION Attribute*, p.466.
- *Step 3:Update the PARENT Attribute*, p.467.
- *Step 4:Create and Install the System Call Handler Function*, p.467.
- *Step 5:Create and Install Header File for System Call Handler*, p.467
- *Step 6:Create and Install Header File for System Call*, p.467.
- *Step 7:Edit and Create Makefiles for System Call Handler*, p.467.
- *Step 8:Define a System Call Group in sysCallUsrNum.def*, p.468.
- *Step 9:Add an Entry for the New System Call to sysCallUsrNum.def*, p.468.
- *Step 10:Define the New System Call in sysCallUserApi.def*, p.468.
- *Step 13:Create a VSB Project and Build VxWorks Libraries With Custom System Call Handlers*, p.469.
- *Step 14:Create a VIP Project and Rebuild VxWorks*, p.469.

19.4 System Call Group Definition File sysCallUsrNum.def

The **syscallUsrNum.def** system call group definition file is used to define a system call group for custom system calls.

Template File

The **syscallUsrNum.def** system call group definition file is created from the template file **syscallUsrNum.def.template**, which is located in *myDir/myName***/vxCustom/myCustomScLayer/custom/syscall_defs.**

System Call Group Number

When you create an entry for your system call group in **syscallUsrNum.def**, you may use one of the groups reserved for Wind River customers (numbers 2 through 7), or define a new system call group.

Wind River system call groups are defined in the **syscallNum.def** file in *installDir***/vxworks-7/pkgs/os/core/syscalls/vxworks/syscall_defs**. The entries reserved for customers use are as follows:

```
SYSCALL_GROUP     SCG_USER0          2
SYSCALL_GROUP     SCG_USER1          3
SYSCALL_GROUP     SCG_USER2          4
SYSCALL_GROUP     SCG_USER3          5
SYSCALL_GROUP     SCG_USER4          6
SYSCALL_GROUP     SCG_USER5          7
```

Wind River recommends that you use one of these groups, as it guarantees they will never be used for VxWorks.

To use one of these system call groups, copy the entry to **syscallUsrNum.def**.

System call group names must be unique. Six system call groups—numbers 2 through 7—are reserved for custom use; all other system call groups are reserved for Wind River use (see *System Call Numbering Rules*, p.477). Contact Wind River if you need to have a group formally added to VxWorks.

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

System Call Group Name

You may use a different system call group name, which means that it would be displayed separately with informational functions (for example, see *19.10 System Call Monitoring And Debugging*, p.474).

If you choose to rename a system call group, define it in **syscallUsrNum.def** using the following syntax:

SYSCALL_GROUP *groupName* *groupNum* *componentNames*

Use a system call group number that is reserved for Wind River customers, and comment out the entry in **syscallNum.def** that uses the same system call group number.

For example, create the following entry in **syscallUsrNum.def**:

```
SYSCALL_GROUP   MY_OWN_SCG     2
```

And comment out the following entry in **syscallNum.def**:

```
SYSCALL_GROUP     SCG_USER0         2
```

For other examples, see the Wind River definitions in *installDir***/vxworks-7/pkgs/os/core/syscalls/vxworks/syscall_defs**, but do not edit them.

System Calls and Components

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

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

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

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

```
SYSCALL_GROUP        SCG_MY_NEW_GROUP    N  INCLUDE_FOO
```

The system calls that are part of the system call group are identified below the **SYSCALL_GROUP** definition line. Up to 64 system calls can be identified within each group.

19.5 System Call API Definition File syscallUsrApi.def

The **syscallUsrApi.def** system call API definition file defines custom system calls.

Syntax

Use the following syntax to define each system call:

INCLUDE *systemCallHandlerheaderFile***.h**

sysCallName numArgs [*argType arg1; argType arg2; argType argN;*] \
[*returnTypeQualifier*] *ComponentName*

Example

```
INCLUDE <mySystemCallHdlr.h>

check_vxworks_version 3 [ int size; const char * buf; \
                           vxWorksVersionInfo * pStruct; ]
```

The header file that is included is the one in *19.18 Example System Call Handler Header File*, p.490.

For other examples, see the VxWorks system call definitions in **syscallApi.def** in *installDir***/vxworks-7/pkgs/os/core/syscalls/vxworks/syscall_defs**.

System Call Handler Header File Location

The header file for the system call handler should be located in the *myDir***/***myName***/vxCustom/***myCustomScLayer***/custom/kernel_h** directory.

For static addition of a system call, the system call handler header file is only necessary when a custom type is being handled by the system call handler. If no user-defined type defined is used, then no **INCLUDE** statement is required with the system call definition.

For more information about the header file itself and its installation, see *Step 5:Create and Install Header File for System Call Handler*, p.467.

System Call Name

The name of the system call used in the system call API definition file (**syscallUsrApi.def**) must match the name used in the system call group definition file (**syscallUsrNum.def**).

Return Type Size Qualifier

The optional qualifier for integer, long, or pointer data type (defined with **int**, **long**, or **ptr**) indicates the size of the system call's return value. If omitted, the default is integer.

CAUTION: The return type size qualifier must be used for 64-bit VxWorks if the return type is long or pointer (with **long** or **ptr**, respectively). Note that this qualifier was introduced with VxWorks 6.9, and cannot be used with earlier releases.

Additional Syntax Requirements

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

When defining the number of arguments, take into consideration any 64-bit arguments and adjust the number accordingly (for issues related to 64-bit arguments, see *System Call Argument Rules*, p.478).

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

More than one component name can be listed. If any of the components is included in the operating system configuration, the system call will be included when the system is built. (For information about custom components, see the *VxWorks Custom Component and CDF Developer's Guide*.)

19.6 Adding System Calls Statically

These steps illustrate the complete process of implementing system calls statically. The examples provided in each step work together cumulatively, and allow you to use the example code.

For the example code, see:

- *19.17 Example System Call Handler*, p.488
- *19.18 Example System Call Handler Header File*, p.490
- *19.19 Example System Call Header File*, p.491.

Step 1: Copy the Default Layer

You will use it to create a variant of the layer. Copy the entire default custom system call layer to another location such as *myDir*/*myName*/**vxCustom**/*myCustomScLayer*.

For example:

```
cd installDir/vxworks-7/pkgs/os/core/syscalls
cp -R custom myDir/myName/vxCustom/myCustomScLayer
```

Step 2: Update the VERSION Attribute

Update the version number in the layer copy by changing the **VERSION** attribute in the **layer.vsbl** file. Do not update the layer name if you want the new layer to replace the default custom system call layer. After the version update, your custom system call layer takes precedence over the default custom system call layer with the same name.

To avoid version number conflicts between your custom system call layer and the Wind River default custom system call layer, start with version 2.0.0.0.

Step 3: Update the PARENT Attribute

If needed, update the parent information of the new layer by modifying the **PARENT** attribute in the **layer.vsbl** file. The **PARENT** attribute is required because the copy of the layer is meant to be located outside of the product installation.

You need to update this attribute if the version of the parent layer changed between the time you copied the default system call layer and when you actually use the copy. For example, **SYSCALLS_1_0_0_0** might have to change to **SYSCALLS_2_0_0_0** if the syscalls layer version changed.

Step 4: Create and Install the System Call Handler Function

Write the system call handler in accordance with the following guidelines:

- *19.14 System Call Handler Requirements*, p. 480
- *19.15 Restrictions on System Call Handlers*, p. 483

For an example, see *19.17 Example System Call Handler*, p. 488. Note that to use the example with the static addition method, you must undefine **DYNAMIC_CUSTOM_SC**.

Add the system call handler to your custom system call layer in

myDir/*myName*/**vxCustom/myCustomScLayer/custom/kernel_src**

Step 5: Create and Install Header File for System Call Handler

Write the header file for the system call handler and add it to the custom system call layer in

myDir/*myName*/**vxCustom/myCustomScLayer/custom/kernel_h**

For an example, see *19.18 Example System Call Handler Header File*, p. 490.

NOTE: If the system call handler only uses parameters of standard types (that is, no user-defined types) the system call handler header file is not required.

Step 6: Create and Install Header File for System Call

Write the header file for the system call itself and add it to the custom system call in

myDir/*myName*/**vxCustom/myCustomScLayer/custom /user_h**

For an example, see *19.19 Example System Call Header File*, p. 491.

Step 7: Edit and Create Makefiles for System Call Handler

In order to build the system call handler, you must create or edit the makefile in

myDir/*myName*/**vxCustom/myCustomScLayer/custom/kernel_src**

As another example, with **mySystemCallHdlr.c** as the system handler source code in the *installDir*/**vxworks-7/pkgs/os/core/syscalls/custom/kernel_src** directory (see *19.17 Example System Call Handler*, p. 488), the makefile would look like this:

```
LIB_BASE_NAME    = MyOwnSc
```

```
include $(WIND_KRNL_MK)/defs.library.mk

OBJS_COMMON=mySystemCallHdlr.o

OBJS        = $(OBJS_COMMON)

include $(WIND_KRNL_MK)/rules.library.mk
```

Step 8: Define a System Call Group in sysCallUsrNum.def

You must define a system call group for your system call (or calls) in the **syscallUsrNum.def** system call group definition file. Create this file from the template file **syscallUsrNum.def.template**, which is located in *myDir*/*myName*/**vxCustom/myCustomScLayer/custom/syscall_defs**.

For detailed information see *19.4 System Call Group Definition File sysCallUsrNum.def*, p.463.

Step 9: Add an Entry for the New System Call to sysCallUsrNum.def

To define a new system call, you must first create an entry for it in the **syscallUsrNum.def** system call group definition file that assigns the system call function a number and associates it with a system call group.

Add the entry for the system call under the appropriate system call group entry, using the following syntax:

sysCallRtnNum sysCallName

Do not use a system call function number that has already been used for the group in question. Reusing an existing number breaks binary compatibility with existing binaries; and all existing applications must be recompiled. System call function numbers do not need to be strictly sequential (that is, there can be gaps in the series for future use).

For example:

```
SYSCALL_GROUP SCG_USER0    2
0 check_vxworks_version
```

For other examples, see the VxWorks system call groups in **syscallNum.def** in *installDir*/**vxworks-7/pkgs/os/core/syscalls/vxworks/syscall_defs**. Do not modify these entries.

For information about the requirements for system call names and their relationship to system call handler names, see *System Call Naming Rules*, p.477. For information about system call numbers and how they are derived from system call group numbers and system call function numbers, see *System Call Numbering Rules*, p.477.

Step 10: Define the New System Call in sysCallUserApi.def

In addition to associating a system call with a system call group, you must also define the system call itself in the **syscallUsrApi.def** system call API definition file.

For detailed information, see *19.5 System Call API Definition File syscallUsrApi.def*, p.465.

Step 11: Trouble-Shoot the Definition Files

The following mistakes are commonly made when editing system call group and system call API definition files, and can confuse the system call infrastructure:

- No space after the opening bracket of an argument list.
- No space before the closing bracket of an argument list.
- No backslash at the end of a line (if the argument list continues onto the next line).
- An empty pair of brackets (which does not enclose any arguments). This causes the generated temporary C file to have a compilation error.

Note that there can be no more than 64 functions in any system call group.

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

Step 12: Set WIND_LAYER_PATHS Environment Variable

Set the **WIND_LAYER_PATHS** environment variable in a shell to the directory holding the variant of the custom system call layer; for example, set **WIND_LAYER_PATHS** to *myDir***/***myName***/vxCustom/***myCustomScLayer*.

Step 13: Create a VSB Project and Build VxWorks Libraries With Custom System Call Handlers

Create and build a VSB project. For example, start **wrtool** and execute the following commands:

```
prj vsb create -bsp qsp_arm -S ./my_qsp_arm_VSB
cd my_qsp_arm_VSB
prj build
```

Step 14: Create a VIP Project and Rebuild VxWorks

Create a VIP project based on your VSB project, configure the VIP as required, and rebuild VxWorks. For example:

```
prj vip create -vsb ./my_qsp_arm_VSB qsp_arm diab ./my_qsp_arm_VIP
cd my_qsp_arm_VIP
prj vip bundle add BUNDLE_STANDALONE_SHELL BUNDLE_RTP_DEVELOP
prj build
```

19.7 Removing Statically-Defined Custom System Calls

Step 1: Delete the appropriate entries from **syscallUsrNum.def** and **syscallUsrApi.def**.

See *19.4 System Call Group Definition File sysCallUsrNum.def*, p.463 and *19.5 System Call API Definition File syscallUsrApi.def*, p.465.

Step 2: Rebuild the VSB project.

Step 3: Rebuild the VIP project.

19.8 Dynamic Addition of System Calls

Custom system calls can be added to VxWorks dynamically as well as statically. With the dynamic method, custom system calls are registered with the system call infrastructure at runtime, rather than when the VxWorks image is built.

To add a system call dynamically, you must add your system call handler and some additional code to VxWorks (by linking it at build time or downloading it at runtime), and then register the system call group at runtime. You do not need to modify the system call definition files, update the generated system call header files used by the system call infrastructure, or rebuild the system's user-side libraries and generate new VxWorks images—as is the case with adding system calls statically (for information in this regard, see *19.3 Static Addition of System Calls*, p.462).

With the dynamic method for adding system calls, you can add your system call code to the runtime system using a downloadable kernel module (DKM) and register the system call group interactively from the shell.

You can also link your system call code with VxWorks, and either configure the system to call the registration function at boot time or implement other code to register the system call group on demand.

Overview of Adding System Calls Dynamically

The basic steps required to add a new system call to VxWorks are as follows:

- *Step 1:Create the System Call Handler*, p.471.
- *Step 2:Add System Call Handler Function Table*, p.471.
- *Step 3:Add System Call Handler Argument Structure*, p.472
- *Step 4:Select a System Call Group Number*, p.472
- *Step 5:Create System Call Handler Group Registration Code*, p.473
- *Step 6:Build the System Call Handler*, p.473
- *Step 7:Register the System Call Group*, p.473

19.9 Adding System Calls Dynamically

These steps illustrate the complete process of implementing system calls dynamically. The examples provided in each step work together cumulatively, and allow you to use the example code.

See *19.17 Example System Call Handler*, p.488 and *19.18 Example System Call Handler Header File*, p.490.

Step 1: Create the System Call Handler

Write the system call handler in accordance with the following guidelines:

- *19.14 System Call Handler Requirements*, p.480
- *19.15 Restrictions on System Call Handlers*, p.483

For an example, see *19.17 Example System Call Handler*, p.488 and *19.18 Example System Call Handler Header File*, p.490. Note that to use this code, the system call handler header file should be located with the system call handler C file.

For information about the requirements for system call handler names and their relationship to system call names, see *System Call Handler Naming Rules*, p.480.

Step 2: Add System Call Handler Function Table

In order to register a system call with VxWorks dynamically, a system call handler function table must be included in the system call handler file. The table is used to register each of the system call handler functions with the runtime system call infrastructure. Each entry in the table consists of one **SYSCALL_DESC_ENTRY( )** macro call for each system call handler function.

The following definition is used in *19.17 Example System Call Handler*, p.488, for one system call handler:

```
#define NUM_RTN       1
LOCAL _WRS_DATA_ALIGN_BYTES(16) SYSCALL_RTN_TBL_ENTRY pRtnTbl [NUM_RTN] =
    {
    SYSCALL_DESC_ENTRY (check_vxworks_versionSc, "check_vxworks_version", 3)
    };
```

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

The arguments to the **SYSCALL_DESC_ENTRY( )** macro are as follows:

- A pointer to the system call handler function.
- A pointer to the system call name—which is displayed with **syscallShow( )**.
- The number of arguments to the system call (the maximum being 8).

To add multiple system call handlers, the system call handler function table would look something like this example:

```
#define NUM_RTN       3
LOCAL _WRS_DATA_ALIGN_BYTES(16) SYSCALL_RTN_TBL_ENTRY testScRtnTbl [NUM_RTN] =
    {
    SYSCALL_DESC_ENTRY (check_vxworks_versionSc, "check_vxworks_version", 3),
    SYSCALL_DESC_ENTRY (fooTestSc, "fooTest", 2),
    SYSCALL_DESC_ENTRY (barTestSc, "barTest", 5)
};
```

Note that while you have defined the system call name in the system call handler function table, you do not use the *name* itself to make a system call at runtime. Instead, you must use the system call group number and the system call function number—which is the appropriate index in the system call handler function table—with **SYSCALL_NUMBER( )** to calculate the system call number. You then use the system call number with **syscall( )** to make the system call (see

19.12 Custom System Calls Testing from an RTP Application, p.476). The only use of the system call name is with the information displayed by **syscallShow( )** see *19.10 System Call Monitoring And Debugging*, p.474).

Step 3: Add System Call Handler Argument Structure

To add a system call to VxWorks dynamically, the system call handler argument structure must be defined in the header file for the system call handler, since it is not automatically generated by the system call infrastructure (as is the case with static addition).

For example, the header file provided in *19.18 Example System Call Handler Header File*, p.490 defines the structure **check_vxworks_versionScArgs** as follows:

```
struct check_vxworks_versionScArgs
    {
    int size;
    const char * buf;
    vxWorksVersionInfo * pStruct;
    };
```

Note that this definition works for 32-bit VxWorks and architectures with a register size of 4 bytes. For 64-bit VxWorks and architectures with a register size of 8 bytes, the **int size** field requires 32-bit padding, either before or after the field (depending on the architecture's endianness). The following example takes this into account:

```
#if (_WRS_INT_REGISTER_SIZE == 8) || defined (_WRS_CONFIG_LP64)
#if (_BYTE_ORDER == _BIG_ENDIAN)
#define _SC_32BIT_ARG_PAD_FIRST         UINT32 padding;
#define _SC_32BIT_ARG_PAD_END
#else
#define _SC_32BIT_ARG_PAD_FIRST
#define _SC_32BIT_ARG_PAD_END           UINT32 padding;
#endif  /* _BYTE_ORDER */
#else
#define _SC_32BIT_ARG_PAD_FIRST
#define _SC_32BIT_ARG_PAD_END
#endif  /* _WRS_INT_REGISTER_SIZE */
struct check_vxworks_versionScArgs
    {
    _SC_32BIT_ARG_PAD_FIRST
    int size;
    _SC_32BIT_ARG_PAD_END
    const char * buf;
    vxWorksVersionInfo * pStruct;
    };
```

The header file in *19.18 Example System Call Handler Header File*, p.490 accommodates both of these cases with conditional code.

Step 4: Select a System Call Group Number

To register your system call with VxWorks, you must select a *system call group number*.

The system call group number for the system call is used as the first parameter of **syscallGroupRegister( )**, which you call at runtime to register the system call group. Do not use a system call group number that is already in use by the system. This means that you should not use any system call group number that has system calls associated with it.

Wind River system call groups are defined in the **syscallNum.def** file in *installDir***/vxworks-7/pkgs/os/core/syscalls/vxworks/syscall_defs**. The numbers reserved for customers use are 2 through 7, defined as follows:

```
SYSCALL_GROUP    SCG_USER0        2
SYSCALL_GROUP     SCG_USER1          3
SYSCALL_GROUP     SCG_USER2          4
SYSCALL_GROUP     SCG_USER3          5
SYSCALL_GROUP     SCG_USER4          6
SYSCALL_GROUP     SCG_USER5          7
```

Wind River recommends that you use one of these numbers, as it guarantees they will never be used for VxWorks, and by default they do not have any system calls associated with them.

You should also be sure that the group number you choose has not been used for static addition of custom system calls for your system (in your development environment). By convention the system call group would be redefined in **syscallUsrNum.def**. For more information in this regard, see *19.3 Static Addition of System Calls*, p.462.

Note that for dynamically added system calls, the value for the group number is important, but not its name. For example, in *19.16 Custom System Call Documentation*, p.487, this system call handler uses a **MY_OWN_SCG** macro for the group number when dynamically adding system calls. The value for this macro is defined as 2, which is the same value used by the **SCG_USER0** system call group. In contrast, for statically added system calls, the name of the system call group and the value of the group number typically matter.

Step 5: Create System Call Handler Group Registration Code

If you choose to write a code to register your system call handler group rather than registering it from the shell directly with **syscallGroupRegister( )**, you can add code to your system call handler file or create a separate module.

For an example, see *19.17 Example System Call Handler*, p.488. For information about **syscallGroupRegister( )**, see the API reference. For information about using **usrAppInit( )** to call a routine at boot time, see *3.12 Automatic Execution of RTP Applications*, p.35.

Step 6: Build the System Call Handler

Create a DKM project containing the system call handler and registration code as you would for any kernel module. The DKM project must based on your VSB project. Then build the project.

You may link your module with VxWorks or download it to the system, depending on your requirements.

Step 7: Register the System Call Group

Boot your system.

If you did not link your system call handler module (and registration module, if separate) with VxWorks, download it to the target.

If you wrote a registration function, call it from the shell. If not, use **syscallGroupRegister( )** directly. For information about **syscallGroupRegister( )**, see the API reference.

Note that if you choose to link your code with VxWorks, you can also configure the system to call the registration function at boot time. For information in this regard, see *1.8 Using userAppInit( ) to Run Applications Automatically*, p.11.

The following code snippet for system call registration is part of the system call handler provided in *19.17 Example System Call Handler*, p.488:

```
#define MY_OWN_SCG    2
#define NUM_RTN    1

/* other code goes here */

void register_my_system_call (void)
    {
    if (syscallGroupRegister (MY_OWN_SCG, "myOwnScGroup", NUM_RTN,
        pRtnTbl, FALSE) == ERROR)
        {
        printf ("syscallGroupRegister() failed. Errno = %#x\n", errnoGet());
        return;
        }
    }
```

In this example, the simple **register_my_system_call( )** function is used to make the **syscallGroupRegister( )** call with the appropriate values. It is important to check the return value from **syscallGroupRegister( )** and print an error message if an error is returned. See the API reference for **syscallGroupRegister( )** for more information.

After you have built the system call handler and the registration function, you can download the module from the VxWorks kernel shell or Workbench, register the system call group, and check that registration has been successful. The following steps illustrate use of the registration code from *19.17 Example System Call Handler*, p.488:

Step 1: Download the module to the target system from Workbench or from the kernel shell.

From the kernel shell (using the C interpreter), for example, the module would be loaded as follows:

```
-> ld < mySystemCallHndlr.o
```

Step 2: Register the new handlers with the system call infrastructure before any system calls are routed to your new handlers:

```
-> register_my_system_call
```

Step 3: Verify that the group is registered by running **syscallShow( )** from the shell.

For more information, see *19.10 System Call Monitoring And Debugging*, p.474.

The system call infrastructure is now ready to route system calls to the newly installed handlers. See *19.11 Custom System Call Optimization*, p.475.

19.10 System Call Monitoring And Debugging

This section discusses using **syscallShow( )**, **syscallMonitor( )**, and hook functions for obtaining information about, and debugging, system calls.

Using syscallShow( )

You can display information about the system calls currently available by calling **syscallShow( )** from the shell (if VxWorks is configured with **INCLUDE_SHOW_ROUTINES**). Detailed information is provided when you use the system call group number for the first parameter and **1** for the second. For example:

```
-> syscallShow
Group Name            GroupNo   NumRtns      Rtn Tbl Addr
--------------------   -------   -------   ------------------
myOwnScGroup            2         1       0x0000000010810010
STANDARDGroup           8         64      0x00000000101921e0
VXWORKSGroup            9         58      0x00000000101925e0
value = 0 = 0x0
-> syscallShow 2,1
System Call Group name: myOwnScGroup
Group Number          : 2

Routines provided     :
Rtn#   Name                          Address          # Arguments
----   ----------------------   ------------------   -----------
0      check_vxworks_version     0x0000000010800000        3
value = 61 = 0x3d = '='
->
```

Note that in this example, the group name **myOwnScGroup** is the one used when the system call was registered dynamically.

For more information, see the **syscallShow( )** entry in the *VxWorks Kernel API Reference*.

Using syscallMonitor( )

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

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

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

For more information, see the **syscallMonitor( )** entry in the *VxWorks Kernel API Reference*.

Using Hook Functions

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

19.11 Custom System Call Optimization

Using system call hook functions for monitoring system calls creates some overhead in the operation of system calls.

If system call hook functions are not required or used in a deployed system, set the **SYSCALL_HOOK_TBL_SIZE** configuration parameter to zero. Note that this also disables the **syscallMonitor( )** facility.

For information about using system call hook functions and **syscallMonitor( )** during development, see *19.10 System Call Monitoring And Debugging*, p. 474.

19.12 Custom System Calls Testing from an RTP Application

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

How you make the system call depends on how it has been added to VxWorks:

- For a system call that has been added statically, simply make the system call as you would for any standard VxWorks system call.
- For a system call that has been added dynamically, you must use **syscall( )**, which takes the system call number (and not its name) as a parameter. See *Testing a Dynamically-Added System Call*, p. 476.

For an example of RTP application code that implements both of these methods, see *19.20 Example RTP Application Using Custom System Calls*, p. 492.

Testing a Dynamically-Added System Call

In order to use a system call that you have added dynamically, you must make the call by way of **syscall( )**, which takes the system call number as on of its parameters, and not the name of the system call.

To calculate the system call number, use the **SYSCALL_NUMBER( )** utility macro (defined in **syscall.h**). The macro takes the system call group number and system call function number for parameters:

- The system call group number is defined with the **syscallGroupRegister( )** call.
- The system call function number is the system call handler function table index for the function.

For more information in this regard, see *Step 2:Add System Call Handler Function Table*, p. 471, *Step 4:Select a System Call Group Number*, p. 472, and *Step 5:Create System Call Handler Group Registration Code*, p. 473.

For example, to call **check_vxworks_version( )** from an RTP application, you would use code like this:

```
typedef struct
    {
    unsigned int major;
    unsigned int minor;
    unsigned int maint;
    } vxWorksVersionInfo;

int check_vxworks_version
    (
    int size,
    const char * buf,
    vxWorksVersionInfo * pStruct
    )
    {
    return syscall (arg1, arg2, arg3, 0, 0, 0, 0, 0,
                    SYSCALL_NUMBER(2,0));
    }
```

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

For the complete RTP code example, see *19.20 Example RTP Application Using Custom System Calls*, p. 492.

19.13 System Call Requirements

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

System Call Naming Rules

For the static method of adding system calls, there is no particular requirement for a system call name, except that it be unique. The names of various elements associated with a system call must, however, derive their names from that of the system call itself. It is important to adhere to this convention in order to avoid compilation errors when using the automated mechanisms provided for adding system calls. See Table 19-2.

For the dynamic method of adding system calls, the same naming convention should be used as for the static method. While there is not *requirement* that the system call handler have the same base name as the system call for the dynamic addition method—because the system call number rather than the system call name is used with **syscall( )** to make the system call—it is preferable for logical consistency and the clarity of your code.

Table 19-2 **System Call Naming Conventions**

Element	Naming Convention
system call	*sysCallName***( )**
system call handler function	*sysCallName***Sc( )**
system call argument structure	*sysCallName***ScArgs**

System Call Numbering Rules

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

A system call number is produced by the concatenation of the following two numbers:

- the system call group number
- the system call function number (within the system call group)

You define the system call group number and system call function number for each of your custom system calls (the method is different for static and dynamic addition of system calls). For the static addition method, the system call infrastructure automatically performs the concatenation that produces the system call numbers. For the dynamic addition method, a macro is provided for you to perform this concatenation programmatically.

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

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

System call function numbers must be unique for the group in question. Reusing an existing number breaks binary compatibility with existing binaries—and all existing applications must be recompiled. System call function numbers do not need to be strictly sequential (that is there can be gaps in the series for future use).

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

System Call Argument Rules

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

For information about the relationship between the arguments of the system call and the argument of the system call handler, see *System Call Handler Argument and Argument Structure Rules*, p.480.

Number of Arguments

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

In cases where more than eight arguments are required the arguments should be packed into a structure whose address is the parameter to the system call. When it is not practical to pass the argument's value with such a structure, a user-side library wrapper can be implemented that handles all those arguments with its interface. This wrapper then packages all the argument values into the argument structure of the actual system call and calls that system call in turn.

64-Bit Argument Issues

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

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

It is important for system call developers to take into account the subtleties of 64-bit argument passing on 32-bit systems. The definition of a system call for VxWorks requires identification of how many words are in the argument list, so that the trap handler can transfer the right amount of data from the user-stack to the kernel-stack. Alignment issues may make this less than straightforward.

Consider for example, the following function prototypes:

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

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

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

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

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

System Call Return Value Rules

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

On 32-bit VxWorks, 64-bit return values are not permitted directly, although they may be emulated by using private functions. To do so, a system call must have a name prefixed by an underscore, and it must a pointer to the return value as one of the parameters. For example the function:

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

must have a companion function:

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

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

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

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

(The **get64BitValue( )** function would be written by the user and placed in a user mode library, and the **_get64BitValue( )** function would be generated automatically.)

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

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

19.14 System Call Handler Requirements

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

A custom system call handler typically calls functions in customer-provided code added to the kernel. A custom system call handler may also call allowed public kernel APIs directly.

CAUTION: Not all kernel functions can be used in system call handlers, and there are limitations on how some may be used. See *19.15 Restrictions on System Call Handlers*, p.483.

For example system call handler code, see the following:

- *19.17 Example System Call Handler*, p.488
- *19.18 Example System Call Handler Header File*, p.490

System Call Handler Naming Rules

System call handlers must be named in accordance with the system call naming conventions, which means that they must use the same name as the system call, but with an **Sc** appended. For example, the **foo( )** system call must be serviced by the **fooSc( )** system call handler. For more information, see *System Call Naming Rules*, p.477.

System Call Handler Argument and Argument Structure Rules

All system call handlers take a single argument, which is a pointer to their argument structure.

The system call argument structure must also be named in accordance with the system call naming conventions, which means that they must use the same name as the system call handler, but with **Args** appended. For more information in this regard, see *System Call Naming Rules*, p.477.

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

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

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

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

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

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

For information about system call arguments, see *System Call Argument Rules*, p.478.

System Call Handler Argument Validation Requirements

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

- Bounds-check numerical values.
- Validate memory addresses used for buffers and data structures.

A system call handler is kernel code that executes in supervisor mode. It is important, therefore, that it evaluate the parameters of a system call to ensure that the values are valid. And in the case of memory addresses, the system call handler should ensure that they correspond to addresses within the RTP's memory context—which may include shared data areas and any other memory mappings that the RTP application may have created.

A system call handler receives all of the system call parameters in one structure, which is passed by address (see *System Call Handler Argument and Argument Structure Rules*, p.480). The structure itself can always be trusted because it is created by the system, but the contents of the structure may not because it comes from an application. For example, the **writeScArgs** structure—discussed in *System Call Handler Argument and Argument Structure Rules*, p.480—is guaranteed to be valid but none of its members is guaranteed to be valid.

Validation of Buffers

A system call handler must validate the actual size of a buffer—rather than the maximum size. If an RTP application allocates a buffer of 20 bytes, the system call handler code should validate those 20 bytes of memory. If the system call handler code attempted to validate the maximum size possible for this buffer, for example 128 bytes, the validation would fail because only 20 bytes were allocated.

The best way to provide the buffer size information to the system call handler is to design the system call with a buffer size parameter.

For a string, if the length is not passed as a parameter, you should validate the first byte of the string, and then assume the remaining part of the string is valid. If you attempt to compute the length of the string, you access the buffer, which causes an exception if the buffer is invalid.

The validation of the buffer must also take into account the type of access allowed on the buffer: read only, or write only, or both read and write.

Buffer Validation Example

For the example of the **writeScArgs** structure (which is passed to the system call handler via the **pArgs** parameter; see *System Call Handler Argument and Argument Structure Rules*, p.480), the validation of the **char * buf** member would be as follows:

```
if (pArgs->nbytes > SOME_AGREED_UPON_MAXIMUM_SIZE)
    {
    // error management
    }

if (scMemValidate (pArgs->buf, pArgs->nbytes, SC_PROT_WRITE) == ERROR)
    {
    // error management
    }
```

Validation of Structures

If a system call has a structure pointer argument, the system call handler must validate the size of the entire structure. Use the **sizeof( )** function to determine the size.

In order to ensure safe access to each element of structure validation, the entire structure must be validated first, and then each member.

Note the following with regard to validating individual members:

- **Members with different access types**. If members of the structure have different access types, it is necessary to apply a global read validation, followed by a write validation to each member that is intended to store an information coming back from the system call handler. Note that a global read and write validation will fail if the structure on the user (RTP) side has been declared as a constant.

- **Pointer members**. If the structure has members that are pointers then a validation step must be done for each pointers, using the appropriate access permission. For buffer pointers, a buffer validation must be done for each of them.

- **Structure members**. If the structure has members that are structures then those structures have to be validated as well.

Structure Validation Example

Consider a system call taking a parameter **pMyStruct** with the type defined as follows:

```
struct myStruct
        {
        char * name;
        int nameSize;
        void * pStorage;
        }
```

The validation of this entire structure (which is passed to the system call handler with the **pArgs** parameter) would be accomplished as follows:

```
if (scMemValidate (pArgs->pMyStruct, sizeof pArgs->pMyStruct, SC_PROT_READ) ==
    ERROR)
    {
     // error management
    }

if (scMemValidate (pArgs->pMyStruct->name, pArgs->pMyStruct->nameSize,
    SC_PROT_READ == ERROR)
    {
     // error management
    }

if (scMemValidate (pArgs->pMyStruct->pStorage, sizeof (void *), SC_PROT_WRITE ==
    ERROR)
    {
    // error management
    }
```

For information about pointer validation across system calls, see the **scMemValidate( )** API reference entry.

System Call Handler Error Reporting

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

19.15 Restrictions on System Call Handlers

Not all kernel functions can be used in system call handlers, and there are limitations on how some kernel functions may be used. System call handlers must adhere to the restrictions described in this section.

In addition, keep in mind that a system call handler must be considered trusted kernel code—it is executed by RTP-side tasks promoted to supervisor mode when they crossed the system call boundary. Once in supervisor mode a task has few constraints and can easily alter both the kernel and the calling RTP's environments with unpredictable results.

Restrictions on Kernel Objects and RTP Objects

The object identifiers returned by various VxWorks APIs are valid for the environment in which they are created. Do not attempt to pass them across system call boundaries (from RTPs to system call handlers or the other way around).

Many VxWorks APIs return an object identifier for an *object* that is created when the API is invoked. Examples include **semMCreate( )**, **taskSpawn( )**, and so on. Many of these APIs are provided for both the kernel and user (RTP) environments. The validity of the identifier is, however, strictly limited to the environment in which the call is made. For example, when a kernel-side task creates a mutex semaphore, the **SEM_ID** object identifier that is returned is only valid in the kernel environment, and cannot be used in any RTP environment (an error status is returned). The same is true when an RTP-side task creates a mutex semaphore; the

SEM_ID object identifier that is returned is only valid in the calling RTP's environment, and cannot be used in the kernel environment, or that of any other RTP.

System call handlers may be executed only by RTP-side tasks when they cross the system call boundary. If a system call handler calls an object-creating API, the identifier that is returned is valid only in the kernel environment. Do not pass the identifier back to the RTP when the RTP-side task returns from the system call. Any attempt to use a kernel object identifier in an RTP will lead to the termination of the RTP due to inappropriate privileges.

Similarly, do not use an RTP object identifier as an argument to a custom system call. The system call handler operates in the kernel environment, and the kernel API corresponding to that type of object identifier would return an error.

There should in any case be no need to create an RTP object by way of a custom system call as the RTP environment provides all object creation APIs relevant to that environment.

Kernel Objects Owned by RTPs

A subclass of kernel objects is associated with the context of the task that created them. These objects include tasks, semaphores, watchdogs, memory partitions, message queues, timers, and RTPs. When these kernel objects are created by a system call handler they are, by default, owned by the RTP to which the creating task belongs. (For more information see the **objLib** API reference.)

If kernel objects owned by an RTP are intended for use by kernel-side tasks, use the **objOwnerSet( )** function to change ownership to the kernel. Otherwise, the objects will not be available to by kernel-side tasks when the RTP that owns them terminates, because they are automatically reclaimed (deleted) at that point.

Note that public named objects are not automatically deleted when their creating RTP terminates. Public named objects maintain a reference count and are deleted only when there are no references to them. For more information about public name objects and system call handlers, see *Caveats for Creating Public Named Objects in System Call Handlers*, p. 487.

Caveat About Using Show Functions

Show functions and shell commands cannot be used to display information about RTP-side objects. A number of information retrieval functions such as **semInfoGet( )** and **msgQInfoGet( )** are, however, available in the RTP environment. They can be used programmatically to retrieve RTP-side information about a number of RTP object categories.

Show functions cannot display information about an RTP object if they are passed an RTP object identifier because they execute in the kernel environment. If, for example, debugging code is used in an RTP application to display the identifier of a given object such as a semaphore, then using the RTP object identifier with a show function such as **semShow( )** from the shell causes an error (with an "**Invalid** *objectName* **id**" error message).

Note that some RTP-side object categories have both an RTP-side object and a corresponding kernel-side object. In these cases, only the identifier of the RTP-side object can be used from within an RTP, and only the kernel-side identifier can be used from within the kernel. This means that show functions and shell commands such as **rtpShow( )**, **i( )**, and **w( )** display kernel object identifiers that relate to RTP objects, which may be a cause for confusion. For example, **i( )** displays kernel identifiers for RTP-side tasks, and **w( )** displays kernel identifiers for RTP-side semaphores and message queues.

Other RTP-side object categories—memory partition identifiers for example—do not have a kernel-side counterpart. Furthermore, some RTP-side object categories may internally use other objects that do have a kernel-side counterpart. For example, POSIX threads are based on VxWorks native tasks and thus use a task object internally.

Limitations on APIs Used in System Call Handlers

Some kernel APIs should not be used in system call handlers because they would violate the separation of the kernel and RTP environment, in some cases triggering an error condition (sometimes fatal). Do not use the following APIs in system call handlers:

- all **pthreadLib** functions
- **rtpSpawn( )**
- **rtpDelete( )**
- **rtpTaskKill( )**
- **rtpTaskSigqueue( )**
- **signal( )**
- **sigaction( )**
- **sigvec( )**
- **taskOpen( )**
- **taskCreate( )**
- **taskSpawn( )**
- **taskInit( )**

The following sections explain why these APIs should not be used.

Problems With Task and POSIX Thread Creation in System Call Handlers

Wind River recommends that you do not create tasks or POSIX threads (pthreads) from a system call handler.

When a task or pthread is created in a system call handler, it is owned by the RTP from which the system call was made. The new task (or pthread) is therefore a user-mode task (or pthread), and its entry point must be a function in the RTP's context. If a kernel function is used as entry point, then this newly created user-mode task or pthread will immediately trigger a segmentation fault which results in the deletion (or the suspension, depending on the policy set with the error detection and reporting facility) of the RTP to which the offending task belongs.

In the case of pthreads this immediate failure also results in a memory leak in the kernel heap if the pthread is created as joinable (which is the default), because the pthread control block will not be reclaimed by a joining pthread—that is, a pthread that called **pthread_join( )**, which is usually the creator of the new pthread.

Calling **pthread_exit( )** in a system call handler causes the calling pthread to be immediately terminated in error. None of the usual operations are conducted on exit (cleanup handlers, unblocking of joined pthreads, freeing of pthread control block), as the RTP-side pthread cannot be managed as a kernel-side pthread.

Although it technically is possible to create a task or pthread from a system call handler—the handler must be passed the address of the entry point in the RTP's environment as one of its arguments—there is no reason to implement this method, as tasks and pthreads can simply be created from the RTP environment directly.

Problems with Child RTP Creation in System Call Handlers

If **rtpSpawn( )** is called in a system call handler, the parent RTP cannot know the child RTP's identifier. This may result in a memory leak in the kernel heap when that child RTP terminates (due to the unclaimed zombie process).

Do not call **rtpDelete( )** in a system call handler. Although this is not currently restricted by **rtpDelete( )** itself, an RTP is not allowed to delete another RTP using this API. Wind River Systems reserves the right to enforce this restriction at any time in future releases. In addition, an RTP should not use **rtpDelete( )** directly to exit as this would bypass the cleanup handlers registered with **atexit( )**.

Problems with Signal Management in System Call Handlers

Do not call **signal( )**, **sigaction( )**, or **sigvec( )** in a system call handler. In addition, do not make a call in a system call handler to any kernel API that involves a signal handler (for example, the **timer_connect( )** and **pthread_create( )** functions).

While calling signal functions in a system call handler does not itself generate an error, an RTP-side task that receives the signal would attempt to execute the kernel-side signal handler, which would trigger a fatal error and termination of the RTP due to the violation of the separation of the kernel and RTP environment. While it is technically possible to pass the address of an RTP-side signal handler with **sigaction( )**, **signal( )**, or **sigvec( )**, this would also trigger a fatal error and termination of the RTP because the kernel signal facility has not been designed to manage signals for RTPs.

You cannot send signals to an RTP's task from a system call handler because there is no way for the caller to determine the kernel object identifier that is associated with the RTP's task.

Problems With Using Hook Functions in System Call Handlers

Hook functions may be registered from a system call handler but, unlike hook functions registered from within the RTP, they apply globally instead of being restricted to the RTP. For example, a task creation hook installed in a system call handler will be called for each task created in the system instead of being called only for tasks created in the RTP.

In addition, RTP deletion hooks are subject to all the limitations described in *Restrictions on Kernel Objects and RTP Objects*, p. 483, because the hook functions are invoked when an RTP terminates and execute from a system call handler (invoked by the **exit( )** function). Furthermore the RTP deletion hook functions must not trigger termination of a task, as this would prevent the proper resource reclamation for the RTP resources by prematurely aborting the RTP termination sequence.

Caveats for Creating Public Named Objects in System Call Handlers

Creation of public named objects (except for tasks) is allowed in system call handlers, but is subject to the same restriction as is described in *Restrictions on Kernel Objects and RTP Objects*, p.483. While the object ID cannot be passed back to the calling RTP, the name can be passed back to the calling RTP—where any task may attempt to open that public object using the relevant RTP-side API. This applies to the following APIs:

- **mq_open( )**
- **msgQOpen( )**
- **sdOpen( )**
- **sem_open( )**
- **semOpen( )**
- **timer_open( )**

Limitations on I/O Drivers

The standard way for RTPs to interact with I/O services is to use the **ioctl( )** system call. The **ioctl( )** system call in turn invokes the I/O driver's support function for **ioctl( )**.

When an RTP-side task calls **ioctl( )** and crosses the system call boundary, all the limitations described in *19.15 Restrictions on System Call Handlers*, p.483 apply. However the same I/O drivers can also be invoked by kernel tasks. This may create an issue as to what the I/O driver's support function for **ioctl( )** may or may not do. Use the **IS_KERNEL_TASK( )** boolean macro in I/O driver code when the I/O driver performs operations that would not be allowed for RTP tasks.

Buffer management in drivers should also be carefully considered, as only RTP tasks that crossed the system call boundary may access the memory of buffers allocated from within their home RTP. It is not uncommon for drivers to pass the address of buffers to external agencies which do the actual operation. Usually such external agencies are kernel tasks that may not access memory allocated in the context of RTPs. In this situation a buffer copy into a kernel-side buffer prior to invoking the external agency is required.

19.16 Custom System Call Documentation

As system calls are not functions written in C, the **apigen** documentation generation utility cannot be used to generate API references from source code comments. You can, however, create a function header in a C file that can be read by **apigen**.

The function header for system calls is no different from that for other C functions.

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

```
/***********************************************************************
*
* getpid - Get the process identifier for the calling process.
*
```

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

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

You have two choices for the location of the comments:

- You may add system call function headers to an existing C source file (one that has code for other functions). Be sure that this source file is part of the **DOC_FILES** list in the makefile for that directory. The **apigen** utility will not process it otherwise.
- You may create a C file that contains only function headers and no C code. Such files must be part of the **DOC_FILES** list in the makefile, but not part of the **OBJS** list (because there is no code to compile).

To generate documentation for the custom system call definition layer, make sure that the **DOC_BUILD** element is set to **YES** in the **layer.vsbl** file located in *installDir***/vxworks-7/pkgs/os/core/syscalls/custom**. If the **DOC_BUILD** element is not set in the file, the default is **YES**.

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

19.17 Example System Call Handler

This system call handler code (**mySystemCallHdlr.c**) implements the **check_vxworks_versionSc( )** function for the **check_vxworks_version( )** system call. It illustrates use of parameters and the **scMemValidate( )** function. It can be used with both the static and dynamic methods for adding custom system calls to VxWorks.

To use this example code with the dynamic method, the system call handler header file should be located with the system call handler C file.

Build your DKM project based on the VSB project that you used for configuring and building VxWorks.

To use this example with the static addition method, you must undefine **DYNAMIC_CUSTOM_SC**.

⚠ **CAUTION:** System call handler files must either include **syscallArgs.h** (for static addition method) or **syscallLib.h** (for the dynamic registration method).

```
#define DYNAMIC_CUSTOM_SC

#include <vxWorks.h>
#include <string.h>
#include <errno.h>
#include "mySystemCallHdlr.h"
#ifdef DYNAMIC_CUSTOM_SC
#include <syscallLib.h>
#else
#include <syscallArgs.h>
#endif /* DYNAMIC_CUSTOM_SC */
#include <scMemVal.h>

#ifdef DYNAMIC_CUSTOM_SC
#define MY_OWN_SCG    2
#define NUM_RTN    1

LOCAL _WRS_DATA_ALIGN_BYTES(16) SYSCALL_RTN_TBL_ENTRY pRtnTbl [NUM_RTN] =
    {
    SYSCALL_DESC_ENTRY (check_vxworks_versionSc, "check_vxworks_version", 3)
    };
#endif /* DYNAMIC_CUSTOM_SC */

int check_vxworks_versionSc
    (
    struct check_vxworks_versionScArgs * pArgs /* struct holding sys call's args
*/
    )
    {
    vxWorksVersionInfo * pStruct = pArgs->pStruct;
    char * pBuf = (char *)pArgs->buf;

    /*
     * Validate that the output parameters are within the RTP's context and can
     * be written into.
     */

     if (scMemValidate ((void *)pArgs->pStruct, sizeof (vxWorksVersionInfo),
            SC_PROT_WRITE) == ERROR)
    {
    errno = EINVAL;        /* structure address is not valid */
    return ERROR;
    }

     if (scMemValidate ((void *)pArgs->buf, (size_t)pArgs->size,
            SC_PROT_WRITE) == ERROR)
    {
    errno = EINVAL;        /* buffer address and/or size is not valid */
    return ERROR;
    }

    /*
     * Copy the VxWorks version information in the output parameters.
     */

    pStruct->major = vxWorksVersionMajor;
    pStruct->minor = vxWorksVersionMinor;
    pStruct->maint = vxWorksVersionMaint;

    if ((strlen (creationDate) + 1) > pArgs->size)
    {
    errno = EMSGSIZE;    /* not enough room in provided buffer */
    return ERROR;
    }

    (void)strncpy (pBuf, creationDate, pArgs->size);

    return OK;
    }

#ifdef DYNAMIC_CUSTOM_SC
void register_my_system_call (void)
    {
    if (syscallGroupRegister (MY_OWN_SCG, "myOwnScGroup", NUM_RTN, pRtnTbl,
                 FALSE) == ERROR)
    {
    printf ("syscallGroupRegister() failed. Errno = %#x\n", errnoGet());
    return;
    }
    }
#endif /* DYNAMIC_CUSTOM_SC */
```

19.18 Example System Call Handler Header File

This header file (**mySystemCallHdlr.h**) defines the **check_vxworks_versionSc( )** system call handler. It is designed so that it can be used with both the static and dynamic methods for adding custom system calls to VxWorks.

For static addition of a system call, the system call handler header file must be added to the kernel side of the VxWorks installation—but only if a custom type is being handled by the system call handler. For more information, see *19.3 Static Addition of System Calls*, p.462.

For dynamic addition of a system call, the system call argument structure must also be defined by the user, because it is not defined and generated by the system call build infrastructure. To use the example code with the dynamic method, the system call handler header file should be located with the system call handler C file. For more information, see *19.3 Static Addition of System Calls*, p.462.

```
#ifndef __INCmySystemCallHdlrh
#define __INCmySystemCallHdlrh

#include <vsbConfig.h>

typedef struct
    {
    unsigned int major;
    unsigned int minor;
    unsigned int maint;
    } vxWorksVersionInfo;

#ifdef DYNAMIC_CUSTOM_SC

/*
 * On VxWorks 64-bit, or if the architecture's register size is 8 bytes,
 * the system call handler's argument structure must take into account the
 * possible padding that the compiler will expect for each 32-bit field.
 * Also, depending on the architecture's byte ordering, the padding can come
 * before or after each 32-bit field.
 */

#if (_WRS_INT_REGISTER_SIZE == 8) || defined (_WRS_CONFIG_LP64)
```

```
#if (_BYTE_ORDER == _BIG_ENDIAN)
#define _SC_32BIT_ARG_PAD_FIRST        UINT32 padding;
#define _SC_32BIT_ARG_PAD_END
#else
#define _SC_32BIT_ARG_PAD_FIRST
#define _SC_32BIT_ARG_PAD_END        UINT32 padding;
#endif  /* _BYTE_ORDER */
#else
#define _SC_32BIT_ARG_PAD_FIRST
#define _SC_32BIT_ARG_PAD_END
#endif  /* _WRS_INT_REGISTER_SIZE */

struct check_vxworks_versionScArgs
    {
    _SC_32BIT_ARG_PAD_FIRST
    int size;
    _SC_32BIT_ARG_PAD_END
    const char * buf;
    vxWorksVersionInfo * pStruct;
    };
#endif /* DYNAMIC_CUSTOM_SC */

#ifdef DYNAMIC_CUSTOM_SC
extern int check_vxworks_versionSc (struct check_vxworks_versionScArgs * pArgs);
#endif /* DYNAMIC_CUSTOM_SC */

#endif /* __INCmySystemCallHdlrh */
```

19.19 Example System Call Header File

This header file (**mySystemCall.h**) defines the system call associated with the example system call handler

See *19.17 Example System Call Handler*, p.488.

For static addition of a system call, the system call header file must be added to the user side of the VxWorks installation, as described in *19.3 Static Addition of System Calls*, p.462.

For dynamic addition of a system call, the system call header file is useful only if a custom type is being handled by the system call.

```
#ifndef __INCmySystemCallh
#define __INCmySystemCallh

typedef struct
    {
    unsigned int major;
    unsigned int minor;
    unsigned int maint;
    } vxWorksVersionInfo;

extern int check_vxworks_version  (int size, const char * buf,
                                   vxWorksVersionInfo * pStruct);

#endif /* __INCmySystemCallh */
```

19.20 Example RTP Application Using Custom System Calls

The following RTP application code can be used to exercise the example system call, which can be added to VxWorks either statically or dynamically.

See *19.17 Example System Call Handler*, p.488, *19.3 Static Addition of System Calls*, p.462, and *19.8 Dynamic Addition of System Calls*, p.470.

The **mySystemCall.h** header file that the application includes is provided in *19.19 Example System Call Header File*, p.491. To use this example code with the dynamic method, the header file should be located with the RTP application C file.

Note that how you make a system call depends on how it has been added to VxWorks:

- For a system call that has been added statically, simply make the system call as you would for any standard VxWorks system call.
- For a system call that has been added dynamically, you must use **syscall( )**, which takes the system call number (and not its name) as a parameter.

For more information, see *19.12 Custom System Calls Testing from an RTP Application*, p.476.

RTP Application checkVxWorksVersion.c

```
/* checkVxWorksVersion.c - demonstrate static custom system calls */
/*
This small application demonstrates the use of a custom system call either
in its static or dynamic version. In the static version, the VxWorks image
must be regenerated to include the support for the check_vxworks_version()
system call. In the dynamic version, the DKM mySystemCallHdlr.o must have been
already loaded in the kernel and its registration function,
register_my_system_call(), executed.
*/

#define DYNAMIC_CUSTOM_SC

/* includes */

#include <vxWorks.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "mySystemCall.h"

#ifdef DYNAMIC_CUSTOM_SC
#include <syscall.h>
#else
#endif /* DYNAMIC_CUSTOM_SC */

#ifdef DYNAMIC_CUSTOM_SC
#define CHECK_VXWORKS_VERSION SYSCALL_NUMBER (2, 0)
#endif

/*******************************************************************************
*
* main - main function of the RTP
*
* This program uses the static custom system call check_vxworks_version() to
* get the version information of the VxWorks system on which it executes.
*
*/

void main (void)
    {
    char creationDate [30];
    vxWorksVersionInfo vxVersionInfo;

#ifndef DYNAMIC_CUSTOM_SC
    printf ("Using static system call check_vxworks_version()\n");

    if (check_vxworks_version  (sizeof creationDate, creationDate,
                &vxVersionInfo) != OK)
    {
    printf ("Error while getting VxWorks version information: "
        "errno = %#x\n", errno);
    exit (1);
    }
#else /* DYNAMIC_CUSTOM_SC */
    printf ("Calling system call check_vxworks_version() via syscall()\n");

    if (syscall (sizeof creationDate, (_Vx_usr_arg_t)creationDate,
         (_Vx_usr_arg_t)&vxVersionInfo,0,0,0,0,0,
         CHECK_VXWORKS_VERSION) == ERROR)
    {
    printf ("syscall() returned an error while getting VxWorks version "
        "information: errno = %#x\n", errno);
    exit (1);
    }
#endif /* DYNAMIC_CUSTOM_SC */

    printf ("This program executed on VxWorks %d.%d.%d\n", vxVersionInfo.major,
            vxVersionInfo.minor, vxVersionInfo.maint);
    printf ("VxWorks image created %s\n", creationDate);

    exit (0);
    }
```

Building the Application checkVxWorksVersion.c

Build the RTP project against the VSB project that you used for configuring and building VxWorks.

Running the Application checkVxWorksVersion.vxe

The example below illustrates starting the RTP application from the kernel shell (using the command interpreter). In this case, the system call was added dynamically, which is indicated by the message stating that it was called by way of **syscall( )**.

```
[vxWorks *]# checkVxWorksVersionRTP.vxe
Launching process 'checkVxWorksVersionRTP.vxe' ...
Process 'checkVxWorksVersionRTP.vxe' (process Id = 0xffff800000195000) launched.
Calling system call check_vxworks_version() via syscall()
This program executed on VxWorks 7.0.0
VxWorks image created Nov 18 2013 10:24:46
[vxWorks *]#
```

20
Custom Scheduler

20.1 About the Custom Scheduler Framework

VxWorks provides a scheduler framework that can be used to implement a custom scheduler. Using a custom scheduler requires configuring VxWorks for its use.

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

- The traditional VxWorks scheduler, which provides for priority-based preemptive scheduling, plus a round-robin extension. The traditional scheduler is included in VxWorks by default. For information, see *VxWorks Traditional Scheduler*, p.86.

- The VxWorks POSIX thread scheduler, which provides additional features for pthreads running in processes. For information, see *9.18 POSIX and VxWorks Scheduling*, p.208.

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

In addition, do not make any assumptions about the compatibility of a custom scheduler across different releases of VxWorks, as the scheduler interface may be changed.

NOTE: The scheduler framework is not supported for the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *18. VxWorks SMP* and *18.18 Code Migration for VxWorks SMP*, p.449.

20.2 About the Traditional VxWorks Scheduler

Wind River recommends understanding the key features of the traditional VxWorks scheduler before designing a custom scheduler.

The traditional scheduler has a priority-based preemptive scheduling policy. A round-robin scheduling extension can be enabled with the **kernelTimeSlice( )** function. For more information about these options, see *VxWorks Traditional Scheduler*, p.86.

The traditional VxWorks scheduler is the default, and is included in the system with the **INCLUDE_VX_TRADITIONAL_SCHEDULER** component.

Scheduler Initialization

The kernel scheduler description structure is initialized in the **usrKernelInit( )** function. The following is an example for configuring the VxWorks traditional scheduler:

```
#ifdef INCLUDE_VX_TRADITIONAL_SCHEDULER

    /* install the traditional priority based preemptive scheduler */

#if (VX_TRAD_SCHED_CONSTANT_RDY_Q == TRUE)
    qPriBMapInit (&readyQHead, Q_TRAD_PRI_BMAP, &readyQBMap,
                  VX_TASK_PRIO_SYSTEM_MAX);
#else
    qPriListInit (&readyQHead, Q_TRAD_PRI_LIST, QUEUE_NONINTERRUPTIBLE);
#endif /* VX_TRAD_SCHED_CONSTANT_RDY_Q == TRUE */

#endif /* INCLUDE_VX_TRADITIONAL_SCHEDULER */
```

This code is from **usrKernel.c**.

The **readyQHead** variable represents the head (**Q_HEAD**) of the scheduler task queue, and can be initialized with the user-specified ready queue class for customized kernel schedulers. Typically, this involves initializing the **pFirstNode** and **pQClass** members of the **readyQHead** structure. See *20.3 Code Requirements for a Custom Scheduler*, p.501 for more information how to install custom schedulers.

Multi-way Queue Structure

The VxWorks scheduler data structure consists of **Q_HEAD**, **Q_NODE**, and **Q_CLASS** elements. The type definitions of **Q_HEAD** and **Q_NODE** structures are flexible so that they can be used for different types of ready queues.

The **readyQHead** variable is the head of a so-called *multi-way queue*, and the aforementioned **Q_PRI_BMAP** queue class (found in **qPriBMapLib.c**) is an example that complies with the multi-way queue data structures.

The multi-way queue *head* structure (**Q_HEAD**) is defined in **private/qLibP.h** as follows:

```
typedef struct          /* Q_HEAD */
    {
    Q_NODE  *pFirstNode;    /* first node in queue based on key */
    ULONG qPriv1;           /* use is queue type dependent */
    ULONG qPriv2;           /* use is queue type dependent */
    Q_CLASS *pQClass;       /* pointer to queue class */
    } Q_HEAD;
```

The first field in the **Q_HEAD** contains the highest priority node.

NOTE: Both the **qFirst( )** function and **Q_FIRST( )** macro simply read the first four bytes (or eight bytes for 64-bit VxWorks) of the **Q_HEAD** structure (the **pFirstNode** field) to determine the head of the queue. There is, therefore, no need for a queue-class specific function to determine which node is the head of the queue.

The kernel scheduler performs a **Q_FIRST( )** on **readyQHead** to determine which task should be allocated to the CPU. For the **Q_PRI_BMAP** and **Q_PRI_LIST** queue classes, this represents the highest priority ready task.

The multi-way queue node structure (**Q_NODE**) is also defined in **qLibP.h** as follows:

```
typedef struct          /* Q_NODE */
    {
    ULONG    qPriv1;            /* use is queue type dependent */
    ULONG    qPriv2;            /* use is queue type dependent */
    ULONG    qPriv3;            /* use is queue type dependent */
    ULONG    qPriv4;            /* use is queue type dependent */
    } Q_NODE;
```

Each task control block contains a **Q_NODE** structure for use by a multi-way queue class to manage the set of ready tasks. This same **Q_NODE** is used to manage a task when it is in a pend queue.

Note that a custom implementation of a multi-way queue class may define class-specific **Q_HEAD** and **Q_NODE** structures. The size of the class-specific structures must not exceed 16 bytes (or 32 bytes for 64-bit VxWorks), which is the current size of both the **Q_HEAD** and **Q_NODE** structures.

Q_CLASS Structure

The kernel interacts with a multi-way queue class through a **Q_CLASS** structure. A **Q_CLASS** structure contains function pointers to the class-specific operators. For example, the address of the class specific *put* function is stored in the **putRtn** field. The **Q_CLASS** structure is defined in **qLibP.h** as follows:

```
typedef struct q_class            /* Q_CLASS */
    {
    void     (*putRtn)             /* insert a node into q  */
        (
        Q_HEAD * pQHead,
        Q_NODE * pQNode,
        ULONG    key
        );
    Q_NODE * (*getRtn)            /* return and remove lead node */
        (
        Q_HEAD * pQHead
        );
    STATUS   (*removeRtn)        /* remove function */
        (
        Q_HEAD * pQHead,
        Q_NODE * pQNode
        );
    void     (*resortRtn)        /* resort node to new priority */
        (
        Q_HEAD * pQHead,
        Q_NODE * pQNode,
        ULONG    key
        );
    void     (*advanceRtn)        /* advance queue */
        (
        Q_HEAD * pQHead,
        ULONG     nTicks
        );
    Q_NODE * (*getExpiredRtn)        /* return and remove expired Q_NODE */
        (
        Q_HEAD * pQHead
        );
    ULONG    (*keyRtn)            /* return insertion key of node */
        (
        Q_NODE * pQNode
        );
    int      (*infoRtn)            /* return array of nodes in queue */
        (
        Q_HEAD * pQHead,
        Q_NODE * nodeArray[],
        int      maxNodes
        );
    Q_NODE * (*eachRtn)            /* call a user function for each node */
        (
        Q_HEAD * pQHead,
        QEACH_USR_FUNC routine,
        void *     routineArg
        );
    void     (*restoreRtn)        /* restore a node position in queue */
        (
        Q_HEAD * pQHead,
        Q_NODE * pQNode,
        ULONG    key
        );
    Q_NODE * (*nextRtn)         /* return next node in a queue */
        (
        Q_HEAD * pQHead,
        Q_NODE * pNode
        );
    struct q_class * valid;        /* valid == pointer to queue class */
    } Q_CLASS;
```

NOTE: The **restoreRtn** and **nextRtn** operators are used in VxWorks SMP, which does not support custom schedulers. These operators must therefore be set to **NULL** for a uniprocessor system.

The following operators are not applicable to a queue class that is used to manage the set of ready tasks: **advanceRtn** and **getExpiredRtn**.

As noted above, a custom scheduler may define class-specific **Q_HEAD** and **Q_NODE** structures.

Q_CLASS Operators

This section provides descriptions of each **Q_CLASS** operator that pertains to the management of ready tasks. Each description provides information about when the kernel invokes the operator for managing ready tasks.

Descriptions of the **advanceRtn** and **getExpiredRtn** operators are not provided as they are not applicable to managing the set of ready tasks.

Some **Q_CLASS** operators are invoked within the *kernel context*. The operator description indicate whether the operator is invoked within kernel context or not.

The operators that are invoked within kernel context do not have access to all VxWorks facilities. Table 20-1 lists the functions that are available from within kernel context.

Table 20-1 **Kernel Context Functions**

VxWorks Library	Available Functions
blib	All functions
fppArchLib	**fppSave( )** and **fppRestore( )**
intLib	**intContext( )**, **intCount( )**, **intVecSet( )**, **intVecGet( )**
lstLib, dllLib, sllLib	All functions except *xxx***Create( )** and *xxx***Delete( )**
mathALib	All functions, if **fppSave( )** and **fppRestore( )** are used
rngLib	All functions except **rngCreate( )**
taskLib	**taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**,**taskIsSuspended( )**and **taskTcb( )**
vxAtomicLib	All functions.

WARNING: The use of any VxWorks APIs that are *not* listed Table 20-1 from an operator that is invoked from kernel context results in unpredictable behavior. Typically the target will hang or reboot.

putRtn

Inserts a node into a multi-way queue. The insertion is based on the key and the underlying queue class. The second parameter is the **Q_NODE** structure pointer of the task to be inserted into the queue. Recall that each task control block contains a **Q_NODE** structure for use by a multi-way queue class to manage the set of ready tasks.

The third parameter, (the key), is the task's current priority. Note that a task's current priority may be different than a task's *normal* priority due to the mutex semaphore priority inheritance protocol.

The **pFirstNode** field of the **Q_HEAD** structure must be updated to contain the first node in the queue (if any change has occurred).

The **putRtn** operator is called whenever a task becomes ready; that is, a task is no longer suspended, pended, delayed, or stopped (or a combination thereof).

The VxWorks round-robin policy performs a **removeRtn** operation followed by a **putRtn** when a task has exceeded its time slice. In this case, the task does not change state. However, the expectation is that by performing a **removeRtn** operation followed by a **putRtn** operation, the task appears as the last task in the *list* of tasks with the same priority, if there are any.

Performing a **taskDelay(0)** operation also results in a **removeRtn** operation followed by a **putRtn**. Again, in this case the task does not change state, and the expectation after performing a **removeRtn** operation followed by a **putRtn** operation is that the task appears as the last task in the *list* of tasks with the same priority, if there are any.

This operator is called from within kernel context.

getRtn
Removes and returns the first node in a multi-way queue. Currently the kernel does not utilize this operator.

removeRtn
Removes the specified node from the specified multi-way queue.

The **removeRtn** operator is called whenever a task is no longer ready; that is, it is no longer eligible for execution, since it has become suspended, pended, delayed, or stopped (or a combination thereof).

See the discussion of the **putRtn** operator above for more information about situations in which the kernel performs a **removeRtn** operation followed by a **putRtn** without the task's state actually changing.

This operator is called from within kernel context.

resortRtn
Resorts a node to a new position based on a new key.

The **resortRtn** operator is called whenever a task's priority changes, either due to an explicit priority change with the **taskPrioritySet( )** API, or an implicit priority change due to the mutex semaphore priority inheritance protocol.

The difference between invoking the **resortRtn** operator and a **removeRtn**/**putRtn** combination is that the former operator does not change the position of the task in the *list* of tasks with the same priority (if any) when the priority is the same as the old priority.

This operator is called from within kernel context.

keyRtn
Returns the key of a node currently in a multi-way queue. The **keyType** parameter determines key style on certain queue classes. Currently the kernel does not utilize this operator.

infoRtn
Gathers information about a multi-way queue. The information consists of an array, supplied by the caller, filled with all the node pointers currently in the queue. Currently the kernel does not utilize this operator.

eachRtn
Calls a user-supplied function once for each node in the multi-way queue. The function should be declared as follows:

```
BOOL routine
    (
    Q_NODE *pQNode,      /* pointer to a queue node */
    _Vx_usr_arg_t arg    /* arbitrary user-supplied argument */
    );
```

The user-supplied function should return **TRUE** if **qEach( )** is to continue calling it for each entry, or **FALSE** if it is done, and **qEach( )** can exit.

Currently the kernel does not utilize this operator.

restoreRtn
Inserts a node into a multi-way queue and restores its position at the head of the queue. This operator is not currently utilized in the kernel.

nextRtn
Returns a pointer to the next node in the queue after the given node. This operator is not currently utilized in the kernel.

20.3 Code Requirements for a Custom Scheduler

A custom scheduler must manage the set of tasks that are in the **READY** state; that is, the tasks that are eligible for execution. At a minimum, a custom scheduler must define a ready queue structure for all ready tasks. An optional hook function that is executed at every clock tick can also be specified.

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

Q_HEAD and Q_NODE Structures

A user-specified ready queue class must define class-specific **Q_NODE** and **Q_HEAD** structures. The size of these structures together must not be more than 16 bytes. Together, these structures define a collection of tasks that are available for execution and the functions used to manipulate this entity. The head of the scheduler queue is stored in a variable named **readyQHead**. For more information, see *Multi-way Queue Structure*, p.496.

Q_CLASS Structure and Associated Functions

Users must define a ready-queue class for all **READY** tasks. A set of functions required by the **Q_CLASS** structure must be implemented. For more information about the **Q_CLASS** structure, see *Multi-way Queue Structure*, p.496.

Task Control Block Data

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

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

- If the **qNode** is given, the **TASK_QNODE_TO_PSCHEDINFO( )** macro may be used to get the address of **pSchedInfo**. The file **taskLib.h** provides the definition of this macro. The macro is typically used in the user-defined queue management functions. For example:

```
void customQPut
    (
    CUSTOM_Q_HEAD     *pQHead, /* head of readyQ */
    CUSTOM_NODE       *pQNode, /* mode of insert */
    ULONG              key     /* key for insert */
    )
    {
    void          **ppSchedInfo;

    /* get the address to the pSchedInFo */

    ppSchedInfo = (void **) TASK_QNODE_TO_PSCHEDINFO (pQNode);
    }
```

- If the task ID **tid** is given, the **TASK_SCHED_INFO_SET( )** macro can be used to set the **pSchedInfo** field in the TCB. The macro **TASK_SCHED_INFO_GET( )** can be used for getting the value of **pSchedInfo**. Both macros are defined in **taskUtilLib.h**.

The custom scheduler may use **pSchedInfo** as the pointer to the user-specific data structure for tasks. If so, it must allocate memory for the data structure using a task hook function that calls **malloc( )** or **memalign( )**. This approach, however, makes the task creation process less deterministic.

The memory can also be statically allocated (using global variables) for user-specified storage, and then used it during task initialization.

Tick Hook Function

If a custom scheduler performs operations at each tick interrupt, the **tickAnnounceHookAdd( )** function can be used to register a hook function that is called at each tick interrupt. The hook function must obey the same rules as ISRs, because it runs in interrupt context. Any VxWorks kernel service that should not be called in an interrupt context should not be called in this hook. For information about restrictions on ISRs, see *8.7 About Interrupt Service Routines: ISRs*, p.163.

The following pseudo-code example illustrates hook use:

```
void usrTickHook
    (
    TASK_ID tid    /* task ID */
    )
    {
    update the statistics information if needed;
    update interrupted task's time slice if needed;
    resort the interrupted task location in the ready queue if needed.
    }
```

Custom Round-Robin Scheduling

VxWorks provides a round-robin scheduling policy based on a task hook function. A custom scheduler may use the VxWorks round-robin scheme by incorporating the **kernelRoundRobinHook( )** function in the user-specific tick hook function.

The **kernelRoundRobinHook( )** function places a task at the tail of the task list for its priority in the ready queue, and resets its time slice, if all of the following are true:

- The interrupted task has not locked preemption.
- The interrupted task is still in the **READY** state.
- The interrupted task has consumed its allowed time slice.

To take advantage of the VxWorks's implementation of round-robin scheduling, the **kernelRoundRobinInstall( )** function should be called in the **usrCustomSchedulerInit( )** function to install the **kernelRoundRobinHook( )** call. For more information see *Modify usrCustomSchedulerInit( ) Function*, p.503.

The function **_func_kernelRoundRobinHook( )** can then be called within the user defined hook for the round robin policy to take effect. The **_func_kernelRoundRobinHook( )** takes the task ID (*tid*) of the interrupted task as its argument. The following code example takes advantage of the VxWorks round-robin scheduling scheme:

```
void usrTickHook

    TASK_ID tid     /* task interrupted by tick */
    )
    {
    /* statistic information */

    /* call kernelRoundRobinHook() */

    if (_func_kernelRoundRobinHook != NULL)
        _func_kernelRoundRobinHook (tid);

    /* other work */
    ...
    }
```

If the custom scheduler does not take advantage of the VxWorks round-robin scheduling policy using **kernelRoundRobinHook( )**, the function **kernelTimeSlice( )** must not be used to adjust system time slice nor to enable or disable round-robin scheduling. The **kernelTimeSlice( )** function is used to dynamically enable round robin scheduling and to set the system time slice, or to disable it.

For more information about VxWorks round-robin scheduling, see *Round-Robin Scheduling*, p. 88.

20.4 Configuration Requirements for a Custom Scheduler

To use a custom scheduler with VxWorks, you must configure VxWorks for use with a custom scheduler, modify an initialization function in a configuration file, and link the custom ready queue management code with VxWorks.

Add INCLUDE_CUSTOM_SCHEDULER Component

To enable the custom scheduler framework, VxWorks must be configured with the **INCLUDE_CUSTOM_SCHEDULER** component.

Modify usrCustomSchedulerInit( ) Function

The **usrCustomSchedulerInit( )** function must be modified to specify the custom ready queue structure and any hook functions that are executed at each tick interrupt. The following code illustrates this modification:

```
void usrCustomSchedulerInit (void)
    {
/*
     * Initialize readyQHead with the custom queue initialization function.
     * Typically this involves initializing a Q_HEAD structure with its Q_CLASS
     * pointer, and possibly some internal structures.
     */

    qCustomInit (&readyQHead, &qCustomClass);

    tickAnnounceHookAdd ((FUNCPTR)usrTickHook);
    kernelRoundRobinInstall();
    }
```

The last two lines are optional. The first is used to install the user defined tick hook function (the **usrTickHook** argument is the hook to be called at each tick interrupt.). The second enables the round robin scheduling policy.

The **usrCustomSchedulerInit( )** function is in **usrCustomScheduler.c** file.

For information about the **readyQHead** variable, see *Scheduler Initialization*, p. 496. This variable must be initialized for a custom scheduler.

Link Custom Scheduler Code

There are several ways for users to link the definition and implementation of **Q_NODE**, **Q_HEAD**, and **Q_CLASS** structure to VxWorks. For example, the custom scheduler configuration file **usrCustomerScheduler.c** can be the placeholder for the **Q_NODE** and **Q_HEAD** type definitions and user specified **Q_CLASS** implementation.

Another way is to create a new header file for **Q_NODE** and **Q_HEAD** definitions and a new source file for **Q_CLASS** implementation, and then link the new object file to VxWorks.

21
Porting C Code from 32-Bit to 64-Bit

21.1 About Porting C Code from 32-Bit to 64-Bit

VxWorks for 64-bit hardware platforms is based on the LP64 data model, whereas VxWorks for 32-bit platforms is based on the ILP32 data model. Code that was written for ILP32 must be carefully examined and modified so as to run correctly with the LP64 data model implementation of VxWorks. Most of the issues described in that chapter are common to porting code from 32-to-64 bit platforms for many operating systems.

Adapting your C code to the LP64 data module is not sufficient for porting it to the 64-bit configuration of VxWorks. To do so, you must also adapt your code to the current release of VxWorks.

With some exceptions—which are largely related to the manipulation of 64-bit entities—code that has been successfully ported to 64-bit VxWorks will successfully re-compile for the ILP32 data model and execute on 32-bit VxWorks.

NOTE: For this release, 64-bit VxWorks is only available for the x86-64 architecture. For detailed information about the X86-64 ABI, see the *VxWorks Architecture Supplement* and the *System V Application Binary Interface AMD64 Architecture Processor Supplement*.

VxWorks 32-Bit and 64-Bit Data Models

Prior to VxWorks 6.8, the operating system was designed for use with 32-bit processors, and adhered to the ILP32 data model. Beginning with VxWorks 6.9, the operating system can be used with 32-bit and 64-bit processors, using the ILP32 and the LP64 data models, respectively. The names of the data models refer to the size (32 or 64 bits) of specific data types, where *I* stands for integer, *L* for long, and *P* for pointer. Table 21-1 shows the size of data types (in bits) for each data model.

Table 21-1 **Size of Data Types for ILP32 and LP64VxWorks**

Data Type	ILP32	LP64
char	8	8
short	16	16
int	32	32
long	32	64
pointer	32	64
long long	64	64
float	32	32
double	64	64

long double	for IA: 96
	for others: 64
		128

off_t	for VxWorks: 32 kernel space 64 user space	64
size_t	32	64
ssize_t	32	64
ptrdiff_t	32	64
enum	32	32

The **long long** type comes from C99. In the LP64 data model it does not differ from the **long** data type; however it does in the ILP32 data model.

The **off_t**, **size_t**, **ssize_t**, and **ptrdiff_t** types are POSIX types that represent a file size, an object's unsigned size, a signed number of bytes read or written (-1 represents an error), and the signed arithmetic difference between two pointers (respectively). They have been introduced for the data size neutrality of some API (one does not have to know what size of objects those types represent in order to use them).

For the ILP32 data model the **off_t** type is indicated in the table as having either a 32-bit size or a 64-bit size. This is specific to VxWorks and indicates the size of **off_t** in the kernel (32 bit) environment and in the RTP environment (64 bits already).

The **long double** type size is not defined by the C99 standard other than being at least equal to that of the **double** type. For the ILP32 data model the Wind River Diab Compiler (**diab**) and GNU compiler implement it as a 64-bit type on all architectures except on x86 where it is a 96-bit (12 bytes) long type. For the LP64 data model **gcc** 3.*x* usually implements it as a 64-bit type except on x86, where it is a 128-bit type. The **gcc** 4.*x* compiler usually implements it as a 128-bit type on all architectures.

Why is the Difference in Data Models Significant?

The main difficulty in porting C code that has been written for a 32-bit hardware platform to a 64-bit hardware platform—or in writing code that is portable between both platforms—has to do with accounting for the ways in which differences in the sizes of data types can cause code to malfunction.

In particular, the change in the size of the pointer type is one of the main causes for the difficulty in porting code from ILP32 to LP64. One can no longer make the assumption that an **int** type variable can hold a pointer (or any address for that matter).

⚠ **CAUTION:** For 64-bit VxWorks, due to the larger size of several data types in the LP64 data model, a task is likely to need a larger stack than what is sufficient for 32-bit VxWorks. Typically an increase of 35% to 50% is enough to most applications, although there may be cases in which the stack size must be doubled. Use the **checkStack( )** shell command to determine if your tasks' stacks are too small (that is, the high water mark is too close to the size of the stack).

21.2 Compiler Identification of Porting Issues

The compiler can be used to flush out problems in code that should be corrected for 64-bit systems.

The following options should be used:

- **-march=core2**
- **-m64**
- -mcmodel=kernel
- -mno-red-zone
- -fno-implicit-fp
- -fno-omit-frame-pointer
- -nostdlib
- -fno-builtin

In addition, the following options are useful for identifying code that may not be portable between 32-bit and 64-bit VxWorks:

- **-Wreturn-type**
- **-Wconversion**
- **–Wformat**
- **-Wno-sign-conversion**

Perform the following steps.

1. Compile your source file with the compiler for the 64-bit architecture. Be sure to Use the proper CPU name to select the proper toolchain: **CORE** selects **ccpentium**.

2. Look for the errors and warnings. See *Compiler Errors and Warnings*, p.508.
3. Look for data size logic issues in your code.
4. Adapt your code following the guidelines provided in *21.3 32-bit/64-Bit C Code Portability*, p.508 and check that it now compiles properly with the compiler for the 64-bit architecture.
5. Compile your source file with the CPU type appropriate for the 32-bit architecture (for example, **PENTIUM4** instead of **CORE**).
6. Check that there are no errors or warnings in the output. If there are, then go back to step 4 and find another way of adapting your code.

Compiler Errors and Warnings

The GNU compiler for x86-64 typically issues one of the following warnings to flag type cast inconsistencies, when using only the -Wall option:

```
    warning: cast to pointer from integer of different size
    warning: cast from pointer to integer of different size
```

These are typically issued when **int** type variables are either assigned (with a cast) with the value of pointer or are representing a pointer to a structure. For example:

```
tid = (int)taskIdCurrent;        (where ‘tid’ is an ‘int’)
((ISR_ID)arg)->isrTag = …;    (where ‘arg’ is an ‘int’)
```

By default no other warnings relevant to the LP64 programming environment are issued.

Using the compiler options **-Wreturn-type**, **-Wconversion**, **-Wformat** and **-Wno-sign-conversion** produce additional useful warnings.

The assembler may produce the following error:

```
Error: suffix or operands invalid for `pop'
```

This error is caused by assembly inlines, usually involving macros. This requires either an update of the inline's assembly code or stopping use of those inlines until they are updated.

21.3 32-bit/64-Bit C Code Portability

Writing code that is portable between 32-bit and 64-bit hardware platforms means taking into account the ways in which differences in the ILP32 and LP64 data models might cause code to malfunction. The aim is to produce code that is *data size neutral*. This section provides guidance both for writing portable code and for porting code from 32-bit to 64-bit hardware platforms.

NOTE: The guidelines provided in this section are intended to be a summary of the key issues involved in making C code portable between 32-bit and 64-bit hardware platforms. They do not provide an exhaustive treatment of the topic (as well as code optimization, and so on).

Check int and long Declarations

While both the **int** and **long** types are both 32 bits in the ILP32 data model, the long type is 64 bits in the LP64 model. Code written for 32-bit hardware platforms may well use **int** and **long** types indiscriminately; and **int** variables are often used as general purpose storage entities. The **int** type cannot be used as a generic type in the LP64 programming environment as not all data types or values fit in a 32-bit storage.

To ensure correct behavior on 64-bit platforms, and portability between 32-bit and 64-bit platforms, make sure that all **int** and **long** assignments are correct for the intended use of the variables and for the platform on which they will be used.

The **long double** type is a *problematic type* due to both the architectural and compiler-related variation in its implementation. If this type is used in your code, consider carefully whether your code really needs it. If so, determine what the compiler produces for the architecture in question.

Check All Assignments

While integers and pointers are the same size on a 32-bit hardware platform, they are not the same size on a 64-bit platform. Addresses cannot, therefore, be manipulated through **int** type variables, function parameters, or function return values.

Data truncation problems can occur with assignments, initialization, returns and so on because the **long** type and pointers are not the same size as **int** with the LP64 data model. Note that these problems are not detected by the compiler (for example, while assigning a 64-bit constant to a 32-bit integer triggers a warning, this does not happen when a 64-bit integer variable is assigned to a 32-bit one).

Check for Data Type Promotion

Some of the most difficult issues with adapting code to the LP64 programming environment are related to integer promotion rules and sign extension involved in logic expressions, assignments, and arithmetic operations. For example, integer promotion rules cause silent truncations when an integer type variable is assigned the value of a larger-size integer type variable.

The C standard should be consulted with regard to data type promotion.

Check Use of Pointers

Pointers and the **int** type are not the same size in the LP64 data model. The common practices for the ILP32 programming environment of using **int** variables to store the values of pointers, of casting pointers to **int** for pointer arithmetic, and so on, will cause pointer corruption.

In addition, because LP64 pointers are 64-bits, it is easy to produce a boundary overflow of an array of pointers unless **sizeof( )** is used to determine the **NULL** value at the end of an array

Check use of sizeof( )

Because the **int** and **long** types are different sizes with the LP64 data model, the use of **sizeof( )** should be examined to make sure the operands are used correctly.

Check Use of Type Casting

While it may be tempting to get rid of compiler warnings by way of type casting, this may do more harm than good because it silences the compiler when it would be indicating a genuine problem.

Type casting can lead to value truncation or, worse, memory corruption when a pointer is involved. For example, consider a function with a signature that has been modified to use a **size_t** * instead of an **int** * parameter, and that is conventionally passed the address of variables holding small values by way of this parameter. If those variables are not updated to be of the **size_t** type but instead are left as **int** and type-cast to **size_t** * when their address is passed to the function, then memory corruption ensues when the function attempts to read or write 8 bytes at those addresses and only 4 bytes have been reserved by the caller code.

It is, therefore, much preferable to change the type of the variables involved in warnings rather than use type casting. There are of course situations where type casting is required, but they should carefully be identified as such.

Check Format String Conversions

String conversion formats for **printf( )** and related functions must be used for the data type in question. The **%d** format will not work with a **long** or pointer for the LP64 programming environment. Use the **%ld** format for longs and **%p** for pointers.

Check Use of Constants

Check the use of all constants. Some of the issues you may encounter include the following:

Integer Constants
An un-suffixed integer constant (that is, without s qualifier suffix such as **U** or **L**) is interpreted as the smallest type that can represent the value, starting with the **int** type. For example, the integer constant **0xFFFFFFFF** is an **int**, but the integer constant **0x100000000** is a **long long** for ILP32 or a **long** for LP64.

0xFFFFFFFF Value
In the LP64 data model the value **0xFFFFFFFF** is not equivalent to -1, and is not even negative. This may lead to test logic failing to work as intended. In the same vein, using **0xFFFFFFFF** as a mask value may not work with LP64. Note that adding the **L** suffix to the constant is not going to help in this case, as it will just tell the compiler that the constant is a **long** instead of an **int**—which does not extend the constant, that is, does not make **0xFFFFFFFF** interpreted as **0xFFFFFFFF.FFFFFFFF**.

ERROR Status
The VxWorks **ERROR** return status must now be checked with equality or difference operators only against the **ERROR** or **OK** constants (comparing ERROR against **0xFFFFFFFF** does not work). The *greater-than* and *less-than* operators should not be used with the **ERROR** constant. Furthermore, in order to avoid compilation warnings it may be necessary to cast the **ERROR** macro with the type of the value against which it is being compared if this value is not of the **STATUS** type, is not an **int**, or may not be equivalent to an **int** on 64-bit VxWorks.

Bit Masks

All masks used in logical operations involving the **long** type should be carefully reviewed due to the change in size of the **long** type between ILP32 and LP64.

Magic Numbers

Other constants to pay attention to include the following:

- **4** (the number of bytes in an integer), does not work for address arithmetic in LP64.
- **32** (the number of bits in an integer), does not work for all integer bit manipulation in LP64.
- **0x7fffffff** (the maximum value of a 32-bit signed variable, often used as mask for zeroing of the right-most bit in a 32-bit type).
- **0x80000000** (the minimum value of a 32-bit signed variable, often used as a mask for allocation of the right-most bit in a 32-bit type).

Check Structures and Unions

The size and alignment of structures will be different in the ILP32 and LP64 programming environments if the structures contain elements that are different sizes in the respective data models. According to the C language, a structure is aligned on the same boundary as its most strictly aligned member.

Structures should, therefore, be examined with consideration to changing the order of members so that as many as possible fall on natural alignment. In general, move **long** and pointer members to the beginning of the structure. While explicit padding may sometimes be useful to counteract automatic padding introduced by the compiler, it should generally be avoided in the context of portability because of data bloat.

Unions should be carefully examined to ensure that the size of the members maintains equivalence with the LP64 data model. For example, the union of an **int** type member and a **long** type member works seamlessly in the ILP32 programming environment but is much more error-prone in the LP64 programming environment where one must be careful not to store the 64-bit value in the 32-bit member of the union (which will result in truncation—possibly silent if only standard warning level is used).

Use Function Prototypes

By default C compilers assume that a function returns a value of type **int**. If your code includes a function that returns a long or a pointer—but for which there is no prototype or for which you did not include the appropriate header file—then the compiler generates code that truncates the return value to fit in 32-bit storage.

Create prototypes for your functions and use the prototypes of functions that are employed in your code.

Use Portable Data Types

All APIs with parameters or return values that represent an offset in memory or in a file, or the size of a region of memory or of a file, or the size of a memory buffer or a file should be changed as follows:

- If the parameter or return value represents a size of a file, or the size of a region of a file, or an offset in a file, replace **int** and unsigned **int** with **off_t**.

- If the parameter represents a region of memory or a buffer, replace **int** and unsigned **int** with **size_t**.
- If the original return value represents a size in memory, but can be negative (usually because it can return -1), then the replacement type should be **ssize_t**. If the return value can only be positive then use **size_t**.

As a Last Resort: Use Conditional Compilation Macro

As a last resort—when code must implement logic that is not portable between data models—the **_WRS_CONFIG_LP64** macro can be used for dual definitions, as follows:

```
#ifdef _WRS_CONFIG_LP64

    /* type definition for LP64 here */

#else /

    /* type definition for ILP32 here */
…
#endif /* _WRS_CONFIG_LP64 */
```

Use this macro only when absolutely necessary—you should try to make code work with both the ILP32 and LP64 programming environments. The macro must, however, be used when the code must implement a logic that cannot apply to one or the other of the data models. The typical case is a type definition involving size when the type name (but not the definition, obviously) must be identical for both ILP32 and LP64. The same is true for macros defining values that cannot be identical for both ILP32 and LP64.

The **_WRS_CONFIG_LP64** macro is available when the **vsbConfig.h** header file is included. The **_WRS_CONFIG_LP64 make** variable is also provided for use in makefiles.

22

Kernel to RTP Application Migration

22.1 Kernel to RTP Application Migration Issues

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 an RTP (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.

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 shared-memory.

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
- VxWorks events
- 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 *19. Custom System Calls*.

WARNING: This is the riskiest option as the possibility exists of breaking the protection of either the kernel or your process.

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 Diab Compiler User's Guide*. 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. 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. For Example:

```
__attribute__((constructor(75))) void hardware_init()
```

```
{
...  // hardware initialization code
}
```

- 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 Diab Compiler User's Guide*.
- 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.

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 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.

- Do not use processor-specific features and instructions in application code. This hampers portability.

Note that interrupts cannot be locked from a process (the **intLock( )** function ins not available).

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 functions 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 *POSIX Signal Differences*, p.519.

Watchdogs

The **wdLib** functions cannot be used in user mode. Replace them with POSIX timers from **timerLib** as shown in Table 22-1.

Table 22-1 **Corresponding wdLib and timerLib Functions**

wdCreate( ) Functions	timer_create( ) Functions
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 22-2.

Table 22-2 **Differences Between Watchdogs and POSIX Timers**

VxWorks wdLib	POSIX timerLib
A function 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, or message queues, or shared data. For more information, see *Eliminating Hardware Access*, p.515.

Redirecting I/O

I/O redirection is possible for the whole process, but not for individual tasks in the process.

The functions **ioTaskStdGet( )** and **ioTaskStdSet( )** are not available in user mode. You can use **dup( )** and **dup2( )** instead, but these functions change the file descriptors for the entire process.

The POSIX **dup( )** and **dup2( )** functions 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 22-1 **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 22-2 **VxWorks 7 Method of I/O Redirection**

The process shown in Example 22-1 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);
```

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 function 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 function 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 function 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 function 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.

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 *7.18 Inter-Process Communication With Public Objects*, p. 147.

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.

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 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.518.

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 in RTPs*, p. 131 and *Message Queues Interruptible by Signals in RTPs*, p. 135.

Signal API Behavior

Table 22-3 shows signal APIs that behave differently in the kernel than in a process.

Table 22-3 **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

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.

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 Diab 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 functions, like **ioGlobalStdSet( )**, that are not available in user mode, generates the following message when using the Diab compiler:

```
fileline ioGlobalStdSet  is deprecated  not available in RTP.
```

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.

22.2 Differences in Kernel and RTP APIs

Many RTP APIs are based on VxWorks kernel APIs, 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 functions 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.

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( )**.

taskInit( )
The **taskInit( )** function 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 functions **taskSwitchHookAdd( )** and **taskSwitchHookDelete( )** do not exist in user mode. However, task delete and create hooks are supported in user mode; therefore the functions **taskCreateHookAdd( )**, **taskCreateHookDelete( )**, **taskDeleteHookAdd( )**, and **taskDeleteHookDelete( )** do exist in user mode. For more information, see *APIs that Work Differently in Processes*, p.522.

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.

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.

APIs that Work Differently in Processes

This section highlights a few APIs that work differently in processes. For more information, see *22.1 Kernel to RTP Application Migration Issues*, p.513.

Task Hook Functions

The **taskCreateHookAdd( )**, **taskDeleteHookAdd( )** functions work differently in processes. The kernel versions of these functions 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 Functions and Migration Macro

Several VxWorks functions have different signatures depending on whether they are the kernel or the user-mode (RTP) variants. These functions 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 functions 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.

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.

Other API Differences

The following are differences between using user-mode APIs in processes and using kernel-level APIs:

- There is no way to get the task list for all tasks in a process.
- Show functions are not available from user mode.

Index