VxWorks® 5.5

PROGRAMMER'S GUIDE

WindRiver®

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

VxWorks Programmer's Guide, 5.5

23 Jul 02
Part #: DOC-14617-ZD-00

Contents

1
Introduction

1.1 Overview

This manual describes the VxWorks real-time operating system, and how to use VxWorks facilities in the development of real-time applications. This manual covers the following topics, focusing first on basic product functionality and facilities, then on optional products and technologies:

- basic operating system functionality
- POSIX standard interfaces
- I/O system
- local file systems, including dosFs
- target tools, such as the shell, target-based loader, and target symbol table
- C++ development using GNU and Diab toolchains
- flash memory device interface (TrueFFS)
- COM and DCOM (VxDCOM)
- distributed message queues (VxFusion)
- shared memory objects (VxMP)
- virtual memory interface (VxVMI)

This chapter describes where to find related documentation about VxWorks and the Tornado development environment. In addition, it describes Wind River customer services, and the document conventions followed in this manual.

1.2 Related Documentation Resources

Detailed information about VxWorks libraries and routines is provided in the *VxWorks API Reference*. Information specific to target architectures is provided in VxWorks architecture supplements,[1] and in the online *VxWorks BSP Reference*.

The VxWorks networking facilities are documented in the *VxWorks Network Programmer's Guide.*

For information about migrating applications, BSPs, drivers, and Tornado projects from previous versions of VxWorks and Tornado, see the *Tornado Migration Guide*.

See the following documents for information on installing and using the Tornado development environment:

- The *Tornado Getting Started Guide* provides information about installing the Tornado development environment and associated optional products.
- The *Tornado User's Guide* provides procedural information about setting up the development environment, and about using Tornado tools to develop VxWorks applications. It includes information on configuring VxWorks systems with the various components described in this guide, and on building and running those systems.

For a complete description of Tornado documentation, see the *Tornado Getting Started Guide: Documentation Guide*.

1.3 VxWorks Configuration and Build

This document describes VxWorks features and configuration options; it does not discuss the mechanisms by which VxWorks-based systems are configured and built. The tools and procedures used for configuring and building those applications are described in the *Tornado User's Guide* and the *Tornado User's Reference*.Tornado provides both GUI and command-line tools for configuration and build.

[1] For example, *VxWorks for PowerPC Architecture Supplement*, *VxWorks for Pentium Architecture Supplement*, *VxWorks for MIPS Architecture Supplement*, and *VxWorks for ARM Architecture Supplement*.

NOTE: In this book, as well as in the *VxWorks API Reference*, VxWorks components are identified by the name used in system configuration files, in the form of **INCLUDE_FOO**. Similarly, configuration parameters are identified by their configuration parameter names, such as **NUM_FOO_FILES**.

Component names can be used directly to identify components and configure VxWorks if you work with the command-line configuration tool and the associated configuration files. The same is true for configuration parameters.

If you use the GUI configuration mechanisms in the Tornado IDE, a simple search facility allows you to locate a component in the GUI based on its component name. Once you have located the component, you can also access the component's parameters through the GUI.

1.4 Wind River Coding Conventions

Wind River has its own coding conventions, which can be seen in the examples in the Tornado and VxWorks documentation. These conventions provide the basis for generating the online Tornado and VxWorks API reference documentation from source code. Following these conventions allows you to use the tools shipped with Tornado to generate your own API documentation in HTML format. For more information, see the *Tornado User's Guide: Coding Conventions*.

1.5 Documentation Conventions

This section describes the documentation conventions used in this manual.

Typographical Conventions

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

Table 1-1 **Typographical Conventions**

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

Cross-References

The cross-references that appear in this guide for subroutines, libraries, or tools refer to entries in the *VxWorks API Reference* (for target routines or libraries) or in the in the *Tornado User's Guide* (for host tools). Cross-references to other books are made at the chapter level, and take the form *Book Title: Chapter Name*; for example, *Tornado User's Guide: Workspace*.

For information about how to access online documentation, see the *Tornado Getting Started Guide: Documentation Guide*.

Directory Pathnames

All VxWorks files reside in the **target** directory (and its subdirectories), directly below the base Tornado installation directory. Because the installation directory is determined by the user, the following format is used for pathnames: *installDir*/**target**.

For example, if you install Tornado in **/home/tornado** on a UNIX host, or in **C:\tornado** on a Windows host, the full pathname for the file identified as *installDir***/target/h/vxWorks.h** in this guide would be **/home/tornado/target/h/vxworks.h** or **C:\tornado\target\h\vxWorks.h**, respectively.

For UNIX users, *installDir* is equivalent to the Tornado environment variable **WIND_BASE**.

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

2
Basic OS

2.1 Introduction

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

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

This chapter discusses the tasking facilities, intertask communication, and the interrupt handling facilities that are at the heart of the VxWorks run-time environment. You can also use POSIX real-time extensions with VxWorks. For more information, see *3. POSIX Standard Interfaces*.

2.2 VxWorks Tasks

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

NOTE: The POSIX standard includes the concept of a thread, which is similar to a task, but with some additional features. For details, see *3.4 POSIX Threads*, p.75.

2.2.1 Multitasking

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

A task's context includes:

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

In VxWorks, one important resource that is *not* part of a task's context is memory address space: all code executes in a single common address space. Giving each task its own memory space requires virtual-to-physical memory mapping, which is available only with the optional product VxVMI; for more information, see *12. Virtual Memory Interface*.

2.2.2 Task State Transition

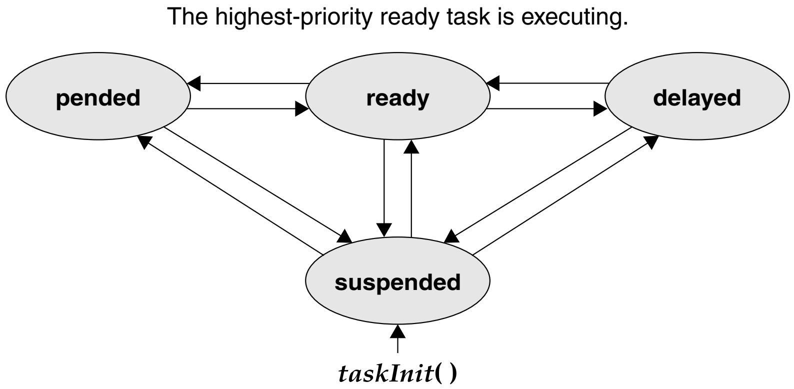
The kernel maintains the current state of each task in the system. A task changes from one state to another as a result of kernel function calls made by the application. When created, tasks enter the *suspended* state. Activation is necessary for a created task to enter the *ready* state. The activation phase is extremely fast, enabling applications to pre-create tasks and activate them in a timely manner. An alternative is the *spawning* primitive, which allows a task to be created and activated with a single function. Tasks can be deleted from any state.

Table 2-1 **Task State Symbols**

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

Table 2-1 describes the *state symbols* that you see when working with Tornado development tools. Figure 2-1 shows the corresponding state diagram of the *wind* kernel states.

Figure 2-1 **Task State Transitions**



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

2.2.3 Wind Task Scheduling

Multitasking requires a scheduling algorithm to allocate the CPU to ready tasks. The default algorithm in *wind* is priority-based preemptive scheduling. You can also select to use round-robin scheduling for your applications. Both algorithms rely on the task's priority. The *wind* kernel has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest. Tasks are assigned a priority when created. You can also change a task's priority level while it is executing by calling **taskPrioritySet( )**. The ability to change task priorities dynamically allows applications to track precedence changes in the real world.

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

POSIX also provides a scheduling interface. For more information, see *3.5 POSIX Scheduling Interface*, p.81.

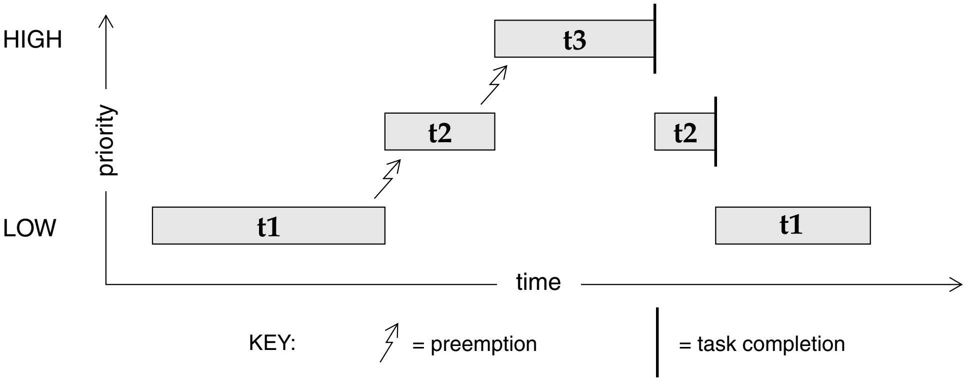
Table 2-2 **Task Scheduler Control Routines**

Call	Description
kernelTimeSlice( )	Controls round-robin scheduling.
taskPrioritySet( )	Changes the priority of a task.
taskLock( )	Disables task rescheduling.
taskUnlock( )	Enables task rescheduling.

Preemptive Priority Scheduling

A *preemptive* priority-based 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 2-2, task **t1** is preempted by higher-priority task **t2**, which in turn is preempted by **t3**. When **t3** completes, **t2** continues executing. When **t2** completes execution, **t1** continues executing.

Figure 2-2 **Priority Preemption**



The disadvantage of this scheduling algorithm 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.

Round-Robin Scheduling

A round-robin scheduling algorithm attempts to share the CPU fairly among all ready tasks of the *same priority*. Round-robin scheduling uses *time slicing* to achieve fair allocation of the CPU to all tasks with the same priority. Each task, in a group of tasks with the same priority, executes for a defined interval or *time slice*.

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

In most systems, it is not necessary to enable round-robin scheduling, the exception being when multiple copies of the same code are to be run, such as in a user interface task.

If round-robin scheduling is enabled, and preemption is enabled for the executing task, the system tick handler increments the task's time-slice count. When the specified time-slice interval is completed, the system tick handler clears the counter and the task is placed at the tail of the list of tasks at its priority level. New tasks joining a given priority group are placed at the tail of the group with their run-time counter initialized to zero.

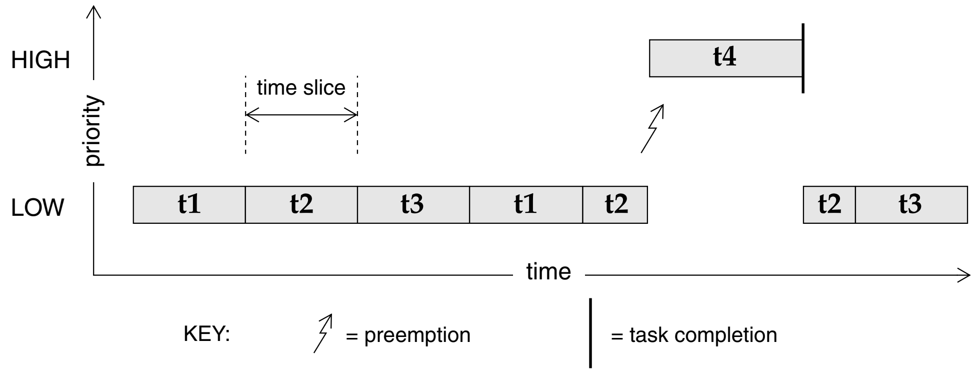
Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.

If a task blocks or is preempted by a higher priority task during its interval, its time-slice count is saved and then restored when the task becomes eligible for execution. In the case of preemption, the task will resume execution once the higher priority task completes, assuming that no other task of a higher priority is ready to run. In the case where the task blocks, it is placed at the tail of the list of tasks at its priority level. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

Time-slice counts are accrued by the task that is executing when a system tick occurs, regardless of whether or not the task has executed for the entire tick interval. Due to preemption by higher priority tasks or ISRs stealing CPU time from the task, it is possible for a task to effectively execute for either more or less total CPU time than its allotted time slice.

Figure 2-3 shows round-robin scheduling for three tasks of the same priority: **t1**, **t2**, and **t3**. Task **t2** is preempted by a higher priority task **t4** but resumes at the count where it left off when **t4** is finished.

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



Preemption Locks

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

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

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

Preemption locks can be used to achieve mutual exclusion; however, keep the duration of preemption locking to a minimum. For more information, see *2.3.2 Mutual Exclusion*, p.33.

A Comparison of taskLock( ) and intLock( )

When using **taskLock( )**, consider that it will not achieve mutual exclusion. Generally, if interrupted by hardware, the system will eventually return to your task. However, if you block, you lose task lockout. Thus, before you return from the routine, **taskUnlock( )** should be called.

When a task is accessing a variable or data structure that is also accessed by an ISR, you can use **intLock( )** to achieve mutual exclusion. Using **intLock( )** makes the operation "atomic" in a single processor environment. It is best if the operation is kept minimal, meaning a few lines of code and no function calls. If the call is too long, it can directly impact interrupt latency and cause the system to become far less deterministic.

Driver Support Task Priority

All application tasks should be priority 100 - 250. However, driver "support" tasks (tasks 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.[1] The system **netTask( )** is at priority 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 Tornado.

2.2.4 Task Control

The following sections give an overview of the basic VxWorks task routines, which are found in the VxWorks library **taskLib**. These routines provide the means for task creation and control, as well as for retrieving information about tasks. See the *VxWorks API Reference* entry for **taskLib** for further information.

For interactive use, you can control VxWorks tasks from the host or target shell; see the *Tornado User's Guide: Shell* and *6. Target Tools* in this manual.

Task Creation and Activation

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

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

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

The **taskSpawn( )** routine creates the new task context, which includes allocating the stack and setting up the task environment to call the main routine (an ordinary

[1] For example, a network interface, an HDLC, and so on.

subroutine) with the specified arguments. The new task begins execution at the entry to the specified routine.

Table 2-3 **Task Creation Routines**

Call	Description
taskSpawn( )	Spawns (creates and activates) a new task.
taskInit( )	Initializes a new task.
taskActivate( )	Activates an initialized task.

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

Task Stack

It is hard to know exactly how much stack space to allocate, without reverse-engineering the system configuration. To help avoid a stack overflow, and task stack corruption, you can take the following approach. When initially allocating the stack, make it much larger than anticipated; for example, from 20KB to up to 100KB, depending upon the type of application. Then, periodically monitor the stack with **checkStack( )**, and if it is safe to make them smaller, modify the size.

Task Names and IDs

When a task is spawned, you can specify an ASCII string of any length to be the task name. VxWorks returns a task ID, which is a 4-byte handle to the task's data structures. Most VxWorks task routines take a task ID as the argument specifying a task. VxWorks uses a convention that a task ID of 0 (zero) always implies the calling task.

VxWorks does not require that task names be unique, but it is recommended that unique names be used in order to avoid confusing the user. Furthermore, to use the Tornado development tools to their best advantage, task names should not conflict with globally visible routine or variable names. To avoid name conflicts, VxWorks uses a convention of prefixing all task names started from the target with the character *t* and task names started from the host with the character *u*.

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

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

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

Call	Description
taskName( )	Gets the task name associated with a task ID.
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.

Task Options

When a task is spawned, you can pass in one or more option parameters, which are listed in Table 2-5. The result is determined by performing a logical OR operation on the specified options.

Table 2-5 **Task Options**

Name	Hex Value	Description
VX_FP_TASK	0x0008	Executes with the floating-point coprocessor.
VX_NO_STACK_FILL	0x0100	Does not fill the stack with 0xee.
VX_PRIVATE_ENV	0x0080	Executes a task with a private environment.
VX_UNBREAKABLE	0x0002	Disables breakpoints for the task.
VX_DSP_TASK	0x0200	1 = DSP coprocessor support.
VX_ALTIVEC_TASK	0x0400	1 = ALTIVEC coprocessor support.

You must include the **VX_FP_TASK** option when creating a task that:

- Performs floating-point operations.
- Calls any function that returns a floating-point value.
- Calls any function that takes a floating-point value as an argument.

For example:

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

Some routines perform floating-point operations internally. The VxWorks documentation for each of these routines clearly states the need to use the **VX_FP_TASK** option.

After a task is spawned, you can examine or alter task options by using the routines listed in Table 2-6. Currently, only the **VX_UNBREAKABLE** option can be altered.

Table 2-6 **Task Option Routines**

Call	Description
taskOptionsGet( )	Examines task options.
taskOptionsSet( )	Sets task options.

Task Information

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

Table 2-7 **Task Information Routines**

Call	Description
taskIdListGet( )	Fills an array with the IDs of all active tasks.
taskInfoGet( )	Gets information about a task.
taskPriorityGet( )	Examines the priority of a task.
taskRegsGet( )	Examines a task's registers (cannot be used with the current task).

Table 2-7 **Task Information Routines**

Call	Description
taskRegsSet( )	Sets a task's registers (cannot be used with the current task).
taskIsSuspended( )	Checks whether a task is suspended.
taskIsReady( )	Checks whether a task is ready to run.
taskTcb( )	Gets a pointer to a task's control block.

Task Deletion and Deletion Safety

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

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

Call	Description
exit( )	Terminates the calling task and frees memory (task stacks and task control blocks only).*
taskDelete( )	Terminates a specified task and frees memory (task stacks and task control blocks only).*
taskSafe( )	Protects the calling task from deletion.
taskUnsafe( )	Undoes a **taskSafe( )** (makes the calling task available for deletion).

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

⚠ **WARNING:** Make sure that tasks are not deleted at inappropriate times. Before an application deletes a task, the task should release all shared resources that it holds.

Tasks implicitly call **exit( )** if the entry routine specified during task creation returns. A task can kill another task or itself by calling **taskDelete( )**.

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

NOTE: You can use VxWorks events to send an event when a task finishes executing. For more information, see *2.4 VxWorks Events*, p.57.

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

Using **taskSafe( )** to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with **taskSafe( )** is blocked. When finished with its critical resource, the protected task can make itself available for deletion by calling **taskUnsafe( )**, which readies any deleting task. To support nested deletion-safe regions, a count is kept of the number of times **taskSafe( )** and **taskUnsafe( )** are called. Deletion is allowed only when the count is zero, that is, there are as many "unsafes" as "safes." Only the calling task is protected. A task cannot make another task safe or unsafe from deletion.

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

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

Deletion safety is often coupled closely with mutual exclusion, as in this example. For convenience and efficiency, a special kind of semaphore, the *mutual-exclusion semaphore*, offers an option for deletion safety. For more information, see *Mutual-Exclusion Semaphores*, p.40.

Task Control

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

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

Table 2-9 **Task Control Routines**

Call	Description
taskSuspend( )	Suspends a task.
taskResume( )	Resumes a task.
taskRestart( )	Restarts a task.
taskDelay( )	Delays a task; delay units are ticks, resolution in ticks.
nanosleep( )	Delays a task; delay units are nanoseconds, resolution in ticks.

Tasks may require restarting during execution in response to some catastrophic error. The restart mechanism, **taskRestart( )**, recreates a task with the original creation arguments.

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

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

The routine **sysClkRateGet( )** returns the speed of the system clock in ticks per second. Instead of **taskDelay( )**, you can use the POSIX routine **nanosleep( )** to specify a delay directly in time units. Only the units are different; the resolution of both delay routines is the same, and depends on the system clock. For details, see *3.2 POSIX Clocks and Timers*, p.73.

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

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

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

NOTE: ANSI and POSIX APIs are similar.

System clock resolution is typically 60Hz (60 times per second). This is a relatively long time for one clock tick, and would be even at 100Hz or 120Hz. Thus, since periodic delaying is effectively *polling*, you may want to consider using event-driven techniques as an alternative.

2.2.5 Tasking Extensions

To allow additional task-related facilities to be added to the system, VxWorks provides hook routines that allow additional routines to be invoked whenever a task is created, a task context switch occurs, or a task is deleted. There are spare fields in the task control block (TCB) available for application extension of a task's context

These hook routines are listed in Table 2-10; for more information, see the reference entry for **taskHookLib**. .

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

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

When using hook routines, be aware of the following restrictions:

- Task switch hook routines must not assume any VM context is current other than the kernel context (as with ISRs).
- Task switch and swap hooks must not rely on knowledge of the current task or invoke any function that relies on this information; for example, **taskIdSelf( )**.
- A switch or swap hook must not rely on the **taskIdVerify(pOldTcb)** mechanism to determine if the delete hook, if any, has already executed for the self-destructing task case. Instead, some other state information needs to be changed; for example, using a **NULL** pointer in the delete hook to be detected by the switch hook.

The **taskCreate***Action* hook routines execute in the context of the creator task, and any new objects are owned by the creator task's home protection domain, or the creator task itself. It may, therefore, be necessary to assign the ownership of new objects to the task that is created in order to prevent undesirable object reclamation in the event that the creator task terminates.

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

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

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

NOTE: For information about POSIX extensions, see *3. POSIX Standard Interfaces*.

2.2.6 Task Error Status: errno

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

Layered Definitions of errno

In VxWorks, **errno** is simultaneously defined in two different ways. There is, as in ANSI C, an underlying global variable called **errno**, which you can display by name using Tornado development tools; see the *Tornado User's Guide*. However, **errno** is also defined as a macro in **errno.h**; this is the definition visible to all of VxWorks except for one function. The macro is defined as a call to a function

__errno( )that returns the address of the global variable, **errno** (as you might guess, this is the single function that does not itself use the macro definition for **errno**). This subterfuge yields a useful feature: because **__errno( )**is a function, you can place breakpoints on it while debugging, to determine where a particular error occurs.

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

```
errno = someErrorNumber;
```

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

A Separate errno Value for Each Task

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

This is accomplished by saving and restoring **errno** on the interrupt stack as part of the interrupt enter and exit code provided automatically by the kernel (see *2.6.1 Connecting Routines to Interrupts*, p.66). Thus, regardless of the VxWorks context, an error code can be stored or consulted with direct manipulation of the global variable **errno**.

Error Return Convention

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

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

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

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

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

Assignment of Error Status Values

VxWorks **errno** values encode the module that issues 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 reference entry on **errnoLib** for more information about defining and decoding **errno** values with this convention.

2.2.7 Task Exception Handling

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

Tasks can also attach their own handlers for certain hardware exceptions through the *signal* facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. 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 *2.3.7 Signals*, p.55 and in the reference entry for **sigLib**.

2.2.8 Shared Code and Reentrancy

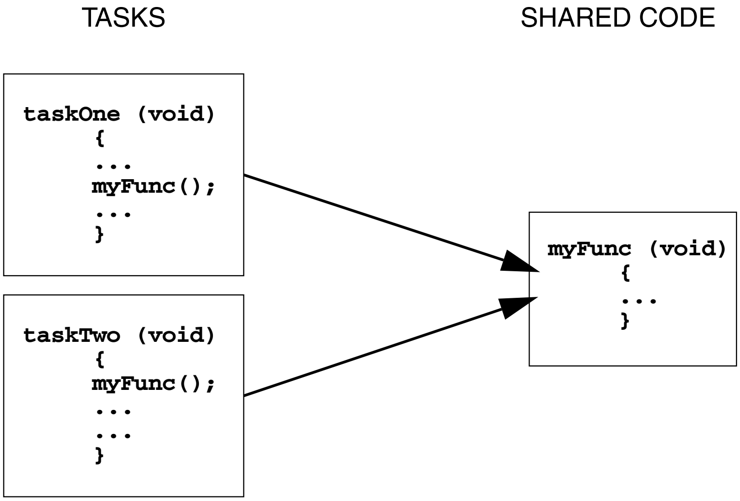
In VxWorks, it is common for a single copy of a subroutine or subroutine library to be invoked by many different tasks. For example, many tasks may call **printf( )**, but there is only a single copy of the subroutine in the system. A single copy of code executed by multiple tasks is called *shared code*.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

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

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

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

Figure 2-4 **Shared Code**



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

The majority of VxWorks routines use the following reentrancy techniques:

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

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

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

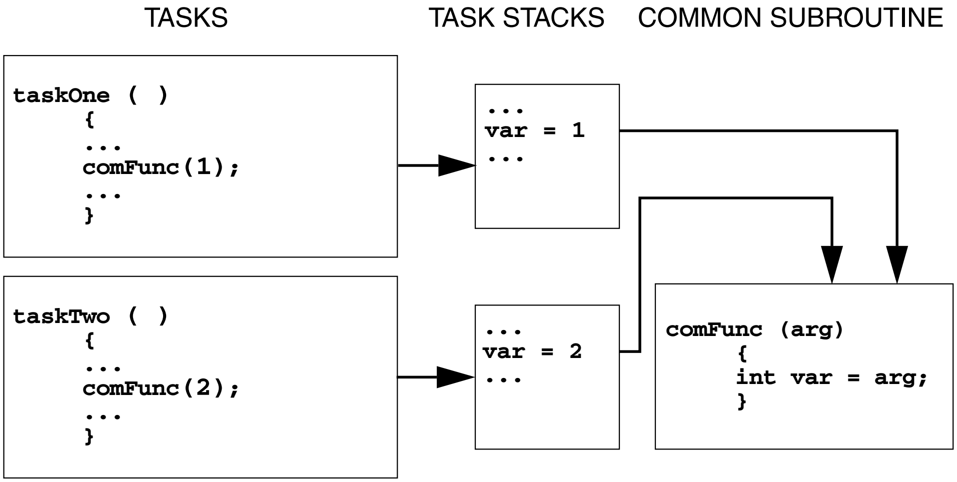
NOTE: **Init( )** functions should be callable multiple times, even if logically they should only be called once. As a rule, functions should avoid **static** variables that keep state information. **Init( )** functions are one exception, where using a **static** variable that returns the success or failure of the original **Init( )** is appropriate.

Dynamic Stack Variables

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

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

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



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

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

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

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

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

Guarded Global and Static Variables

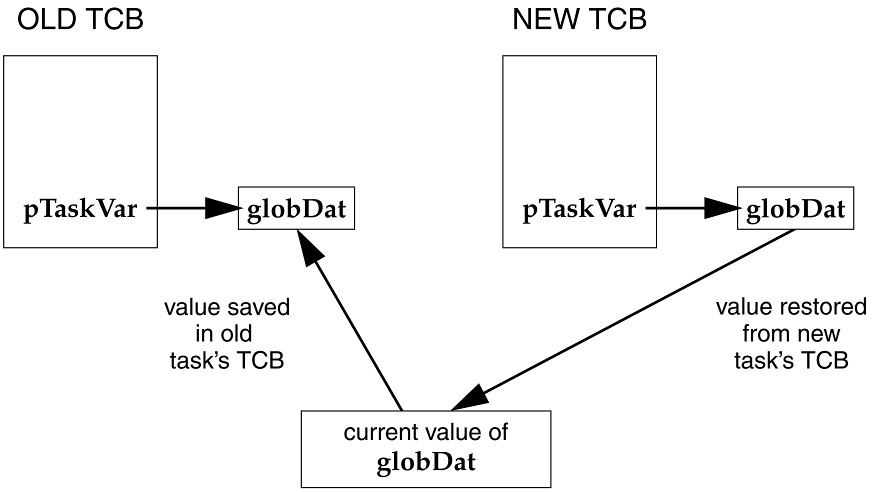
Some libraries encapsulate access to common data. This kind of library requires some caution because the routines are not inherently reentrant. Multiple tasks simultaneously invoking the routines in the library might interfere with access to common variables. Such libraries must be made explicitly reentrant by providing a *mutual-exclusion* mechanism to prohibit tasks from simultaneously executing critical sections of code. The usual mutual-exclusion mechanism is the mutex semaphore facility provided by **semMLib** and described in *Mutual-Exclusion Semaphores*, p.40.

Task Variables

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

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

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



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

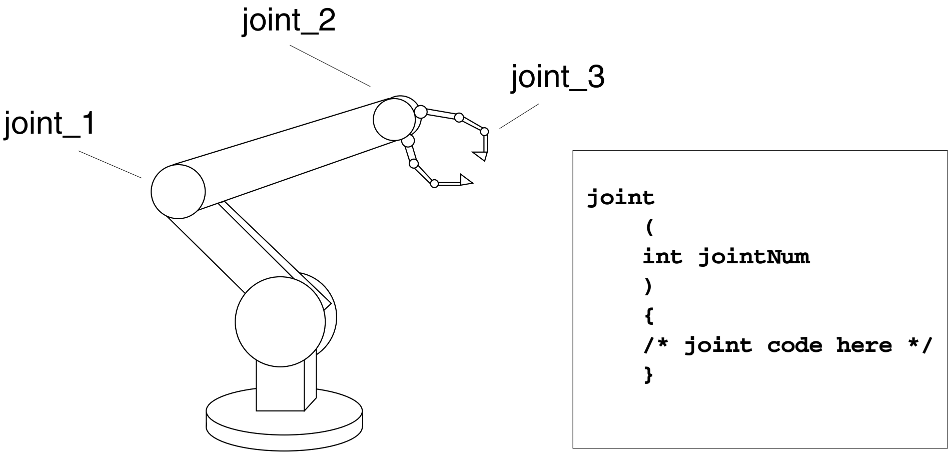
Multiple Tasks with the Same Main Routine

With VxWorks, it is possible to spawn several tasks with the same main routine. Each spawn creates a new task with its own stack and context. Each spawn can also pass the main routine different parameters to the new task. In this case, the same rules of reentrancy described in *Task Variables*, p. 28 apply to the entire task.

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

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

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



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

2.2.9 VxWorks System Tasks

Depending on its configuration, VxWorks may include a variety of system tasks. These are described below.

Root Task: tUsrRoot

The root task is the first task executed by the kernel. The entry point of the root task is **usrRoot( )**in *installDir***/target/config/all/usrConfig.c** and initializes most VxWorks facilities. It spawns such tasks as the logging task, the exception task, the network task, and the **tRlogind** daemon. Normally, the root task terminates and is deleted after all initialization has occurred.

Logging Task: tLogTask

The log task, **tLogTask**, is used by VxWorks modules to log system messages without having to perform I/O in the current task context. For more information, see *4.5.3 Message Logging*, p.122 and the reference entry for **logLib**.

Exception Task: tExcTask

The exception task, **tExcTask**, supports the VxWorks exception handling package by performing functions that cannot occur at interrupt level. It is also used for actions that cannot be performed in the current task's context, such as task suicide. It must have the highest priority in the system. Do not suspend, delete, or change the priority of this task. For more information, see the reference entry for **excLib**.

Network Task: tNetTask

The **tNetTask** daemon handles the task-level functions required by the VxWorks network. Configure VxWorks with the **INCLUDE_NET_LIB** component to spawn the **tNetTask** task.

Target Agent Task: tWdbTask

The target agent task, **tWdbTask**, is created if the target agent is set to run in task mode. It services requests from the Tornado target server; for information about this server, see the *Tornado User's Guide: Overview*. Configure VxWorks with the **INCLUDE_WDB** component to include the target agent.

Tasks for Optional Components

The following VxWorks system tasks are created if their associated configuration constants are defined; for more information, see the *Tornado User's Guide: Configuration and Build*.

tShell

If you have included the target shell in the VxWorks configuration, it is spawned as this task. Any routine or task that is invoked from the target shell, rather than spawned, runs in the **tShell** context. For more information, see *6. Target Tools*. Configure VxWorks with the **INCLUDE_SHELL** component to include the target shell.

tRlogind

If you have included the target shell and the **rlogin** facility in the VxWorks configuration, this daemon allows remote users to log in to VxWorks. It accepts a remote login request from another VxWorks or host system and spawns **tRlogInTask** and **tRlogOutTask**. These tasks exist as long as the remote user is logged on. During the remote session, the shell's (and any other task's) input and output are redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**. For more information, see *4.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)*, p.132 and the reference entry for **ptyDrv**. Configure VxWorks with the **INCLUDE_RLOGIN** component to include the **rlogin** facility.

tTelnetd

If you have included the target shell and the **telnet** facility in the VxWorks configuration, this daemon allows remote users to log in to VxWorks with **telnet**. It accepts a remote login request from another VxWorks or host system and spawns the input task **tTelnetInTask** and output task **tTelnetOutTask**. These tasks exist as long as the remote user is logged on. During the remote session, the shell's (and any other task's) input and output are redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**. See *4.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)*, p.132 and the reference entry for **ptyDrv** for further explanation. Configure VxWorks with the **INCLUDE_TELNET** component to include the telnet facility.

tPortmapd

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

2.3 Intertask Communications

The complement to the multitasking routines described in *2.2 VxWorks Tasks*, p.8 is the intertask communication facilities. These facilities permit independent tasks to coordinate their actions.

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

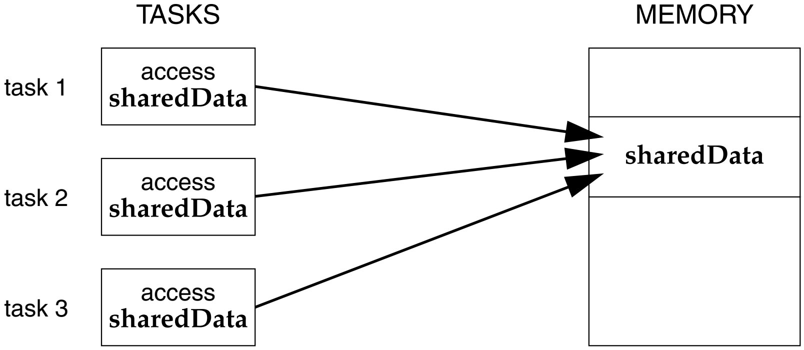
- *Shared memory*, for simple sharing of data.
- *Semaphores*, for basic mutual exclusion and synchronization.
- *Mutexes* and *condition variables* for mutual exclusion and synchronization using POSIX interfaces.
- *Message queues* and *pipes*, for intertask message passing within a CPU.
- *Sockets* and *remote procedure calls*, for network-transparent intertask communication.
- *Signals*, for exception handling.

The optional products VxMP and VxFusion provide for intertask communication between multiple CPUs. See *11. Shared-Memory Objects* and *10. Distributed Message Queues*.

2.3.1 Shared Data Structures

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

Figure 2-8 **Shared Data Structures**



2.3.2 Mutual Exclusion

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

For information about POSIX mutexes, see *3.7 POSIX Mutexes and Condition Variables*, p.92.

Interrupt Locks and Latency

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

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

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

⚠ **WARNING:** Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

Preemptive Locks and Latency

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

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

However, this method can lead to unacceptable real-time response. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region. While this kind of mutual exclusion is simple, if you use it, make sure to keep the duration short. A better mechanism is provided by semaphores, discussed in *2.3.3 Semaphores*, p.34.

⚠ **WARNING:** The critical region code should not block. If it does, preemption could be re-enabled.

2.3.3 Semaphores

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

- For *mutual exclusion* semaphores interlock access to shared resources. They provide mutual exclusion with finer granularity than either interrupt disabling or preemptive locks, discussed in *2.3.2 Mutual Exclusion*, p.33.
- For *synchronization* semaphores coordinate a task's execution with external events.

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

binary
: The fastest, most general-purpose semaphore. Optimized for synchronization or mutual exclusion.

mutual exclusion

A special binary semaphore optimized for problems inherent in mutual exclusion: priority inheritance, deletion safety, and recursion.

counting

Like the binary semaphore, but keeps track of the number of times a semaphore is given. Optimized for guarding multiple instances of a resource.

VxWorks provides not only the Wind semaphores, designed expressly for VxWorks, but also POSIX semaphores, designed for portability. An alternate semaphore library provides the POSIX-compatible semaphore interface; see *3.6 POSIX Semaphores*, p.85.

The semaphores described here are for use on a single CPU. The optional product VxMP provides semaphores that can be used across processors; see *11. Shared-Memory Objects*.

Semaphore Control

Instead of defining a full set of semaphore control routines for each type of semaphore, the Wind semaphores provide a single uniform interface for semaphore control. Only the creation routines are specific to the semaphore type. Table 2-12 lists the semaphore control routines.

Table 2-12 **Semaphore Control Routines**

Call	Description
semBCreate( )	Allocates and initializes a binary semaphore.
semMCreate( )	Allocates and initializes a mutual-exclusion semaphore.
semCCreate( )	Allocates and initializes a counting semaphore.
semDelete( )	Terminates and frees a semaphore.
semTake( )	Takes a semaphore.
semGive( )	Gives a semaphore.
semFlush( )	Unblocks all tasks that are waiting for a semaphore.

The **semBCreate( )**, **semMCreate( )**, and **semCCreate( )** routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by the other semaphore-control routines. When a semaphore is created, the queue type is specified. Tasks pending on a semaphore can be queued in priority order (**SEM_Q_PRIORITY**) or in first-in first-out order (**SEM_Q_FIFO**).

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

Binary Semaphores

The general-purpose binary semaphore is capable of addressing the requirements of both forms of task coordination: mutual exclusion and synchronization. The binary semaphore has the least overhead associated with it, making it particularly applicable to high-performance requirements. The mutual-exclusion semaphore described in *Mutual-Exclusion Semaphores*, p.40 is also a binary semaphore, but it has been optimized to address problems inherent to mutual exclusion. Alternatively, the binary semaphore can be used for mutual exclusion if the advanced features of the mutual-exclusion semaphore are deemed unnecessary.

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

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

Figure 2-9 **Taking a Semaphore**

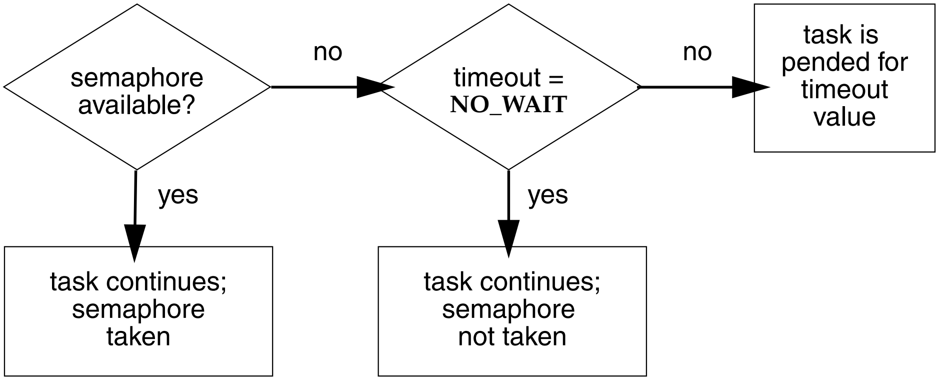
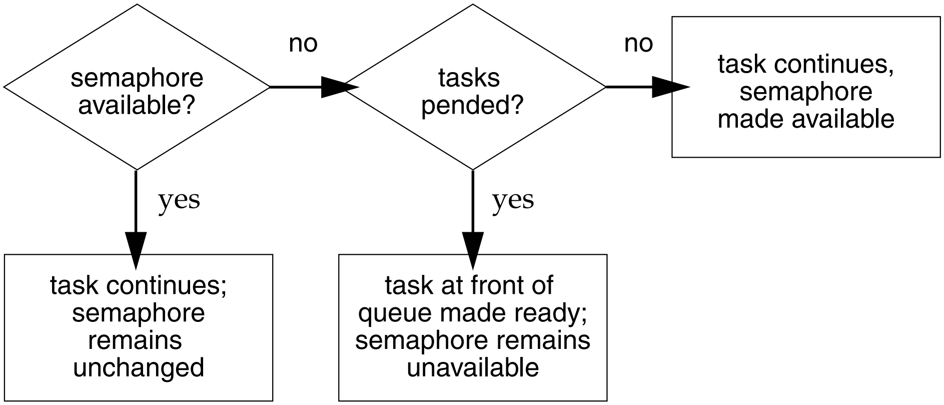

semaphore available?
no
timeout = NO_WAIT
no
task is pended for timeout value
yes
task continues; semaphore taken
yes
task continues; semaphore not taken

Figure 2-10 **Giving a Semaphore**


semaphore available?
no
tasks pended?
no
task continues, semaphore made available
yes
task continues; semaphore remains unchanged
yes
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 (see *2.6 Interrupt Service Code: ISRs*, p.65 for a complete discussion of ISRs). Another task waits for the semaphore by calling **semTake( )**. The waiting task blocks until the event occurs and the semaphore is given.

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

In Example 2-1, the **init( )** routine creates the binary semaphore, attaches an ISR to an event, and spawns a task to process the event. The routine **task1( )** runs until it calls **semTake( )**. It remains blocked at that point until an event causes the ISR to call **semGive( )**. When the ISR completes, **task1( )** executes to process the event. There is an advantage of handling event processing within the context of a dedicated task: less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

Example 2-1 **Using Semaphores for Task Synchronization**

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

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

SEM_ID syncSem;             /* ID of sync semaphore */

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

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

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

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

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

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

Mutual-Exclusion Semaphores

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

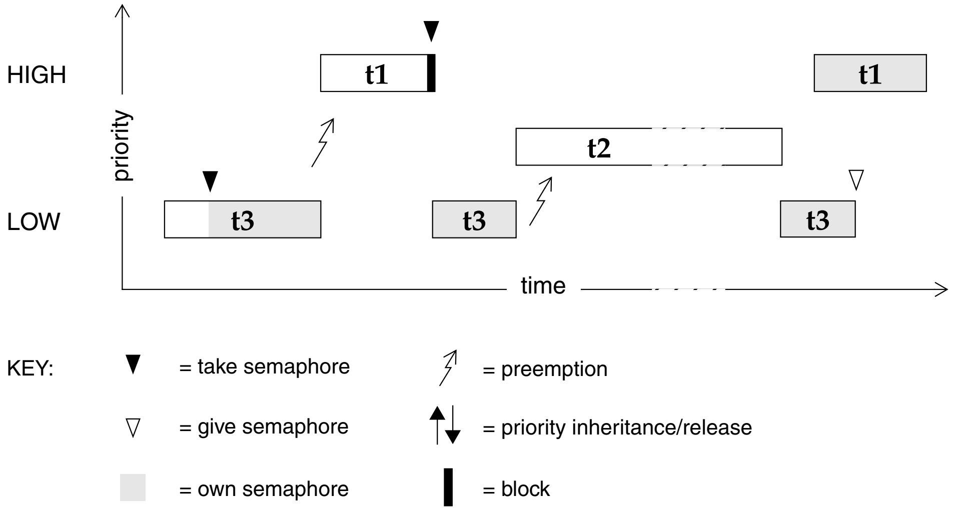
The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

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

Priority Inversion

Figure 2-11 illustrates a situation called priority inversion.

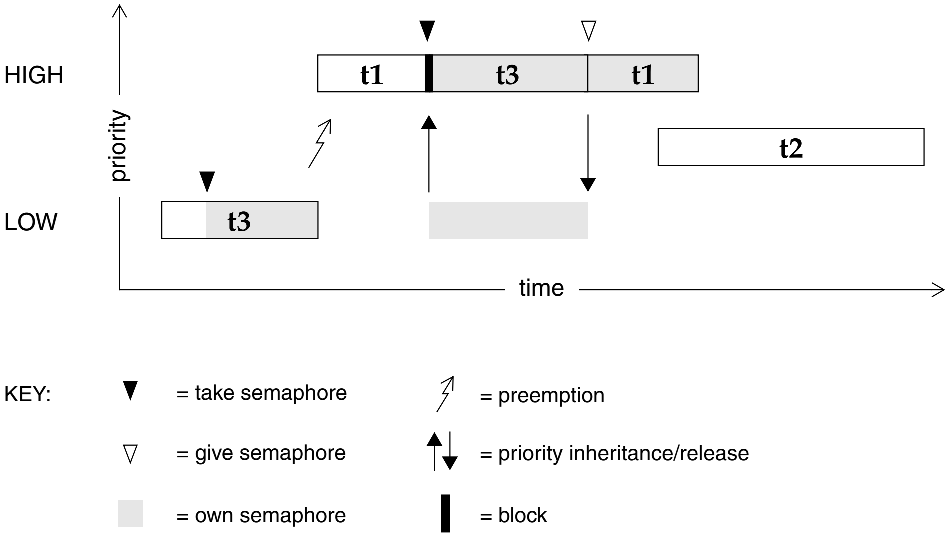
Figure 2-11 **Priority Inversion**



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

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

Figure 2-12 **Priority Inheritance**



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

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

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

Deletion Safety

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

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

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

Recursive Resource Access

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

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

Example 2-2 **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");
    }
```

Counting Semaphores

Counting semaphores are another means to implement task synchronization and mutual exclusion. The counting semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore is given. Every time a semaphore is given, the count is incremented; every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, unlike the binary semaphore, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking. Table 2-13 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Table 2-13 **Counting Semaphore Example**

Semaphore Call	Count after Call	Resulting Behavior
semCCreate( )	3	Semaphore initialized with an initial count of 3.
semTake( )	2	Semaphore taken.

Table 2-13 **Counting Semaphore Example**

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

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

Special Semaphore Options

The uniform Wind semaphore interface includes two special options. These options are not available for the POSIX-compatible semaphores described in *3.6 POSIX Semaphores*, p.85.

Timeouts

As an alternative to blocking until a semaphore becomes available, semaphore take operations can be restricted to a specified period of time. If the semaphore is not taken within that period, the take operation fails.

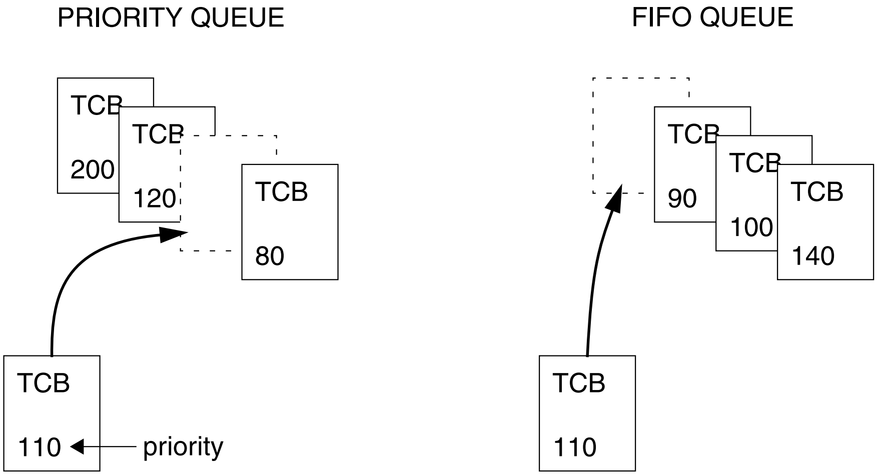
This behavior is controlled by a parameter to **semTake( )** that specifies the amount of time in ticks that the task is willing to wait in the pended state. If the task succeeds in taking the semaphore within the allotted time, **semTake( )** returns **OK**. The **errno** set when a **semTake( )** returns **ERROR** due to timing out before successfully taking the semaphore depends upon the timeout value passed.

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

Queues

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

Figure 2-13 **Task Queue Types**



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

Semaphores and VxWorks Events

This section describes using VxWorks events with semaphores. You can also use VxWorks events with other VxWorks objects. For more information, see *2.4 VxWorks Events*, p.57.

Using Events

A semaphore can send events to a task, if it is requested to do so by the task. To request that a semaphore send events, a task must register with the semaphore using **semEvStart( )**. From that point on, every time the semaphore is released with **semGive( )**, and as long as no other tasks are pending on it, the semaphore sends events to the registered task. To request that the semaphore stop sending events, the registered task calls **semEvStop( )**.

Only one task can be registered with a semaphore at any given time. The events a semaphore sends to a task can be retrieved by the task using routines in **eventLib**. Details on when semaphores send events are documented in the reference entry for **semEvStart( )**.

In some applications, the creator of a semaphore may wish to know when a semaphore failed to send events. Such a scenario can occur if a task registers with a semaphore, and is subsequently deleted before having time to unregister. In this situation, a given operation could cause the semaphore to attempt to send events to the deleted task. Such an attempt would obviously fail. If the semaphore is created with the **SEM_EVENTSEND_ERROR_NOTIFY** option, the given operation returns an error. Otherwise, VxWorks handles the error quietly.

Using **eventReceive( )**, a task may pend on events meant to be sent by a semaphore. If the semaphore is deleted, the task pending on events is returned to the ready state, just like the tasks that may be pending on the semaphore itself.

Existing VxWorks API

The VxWorks events implementation does not propose to keep track of all the resources a task is currently registered with. Therefore, a resource can attempt to send events to a task that no longer exists. For example, a task may be deleted or may self-destruct while still registered with a resource to receive events. This error is detected only when the resource becomes free, and is reported by having **semGive( )** return **ERROR**. However, in this case the error does not mean the semaphore was not given or that the message was not properly delivered. It simply means the resource could not send events to the registered task. This is a different behavior from the one presently in place under VxWorks, however it is the same behavior that exists for pSOS message queues and semaphores.

Performance Impact

When a task is pending for the semaphore, there is no performance impact on **semGive( )**. However, if this is not the case (for example, if the semaphore is free), the call to **semGive( )** takes longer to complete since events may have to be sent to a task. Furthermore, the call may unpend a task waiting for events, which means the caller may be preempted, even if no task is waiting for the semaphore.

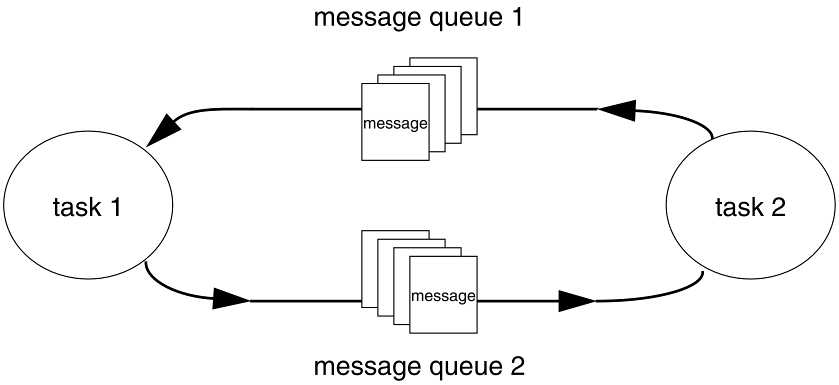
The **semDestroy( )** routine performance is impacted in cases where a task is waiting for events from the semaphore, since the task has to be awakened. Also note that, in this case, events need not be sent.

2.3.4 Message Queues

Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In VxWorks, the primary intertask communication mechanism within a single CPU is *message queues. (*The VxWorks distributed message queue component provides for sharing message queues between processors across any transport media; see *10. Distributed 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 2-14 **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 2-14.

There are two message-queue subroutine libraries in VxWorks. The first of these, **msgQLib**, provides Wind message queues, designed expressly for VxWorks; the second, **mqPxLib**, is compatible with the POSIX standard (1003.1b) for real-time extensions. See *3.5.1 Comparison of POSIX and Wind Scheduling*, p.82 for a discussion of the differences between the two message-queue designs.

Wind Message Queues

Wind message queues are created, used, and deleted with the routines shown in Table 2-14. This library provides messages that are queued in FIFO order, with a single exception: there are two priority levels, and messages marked as high priority are attached to the head of the queue.

Table 2-14 **Wind Message Queue Control**

Call	Description
msgQCreate( )	Allocates and initializes a message queue.
msgQDelete( )	Terminates and frees a message queue.
msgQSend( )	Sends a message to a message queue.
msgQReceive( )	Receives a message from a message queue.

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

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

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

Timeouts

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

Urgent Messages

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

Example 2-3 **Wind Message Queues**

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

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

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

MSG_Q_ID myMsgQId;

task2 (void)
    {
    char msgBuf[MAX_MSG_LEN];

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

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

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

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

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 Wind message queue, the output is sent to the standard output device, and looks like the following:

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

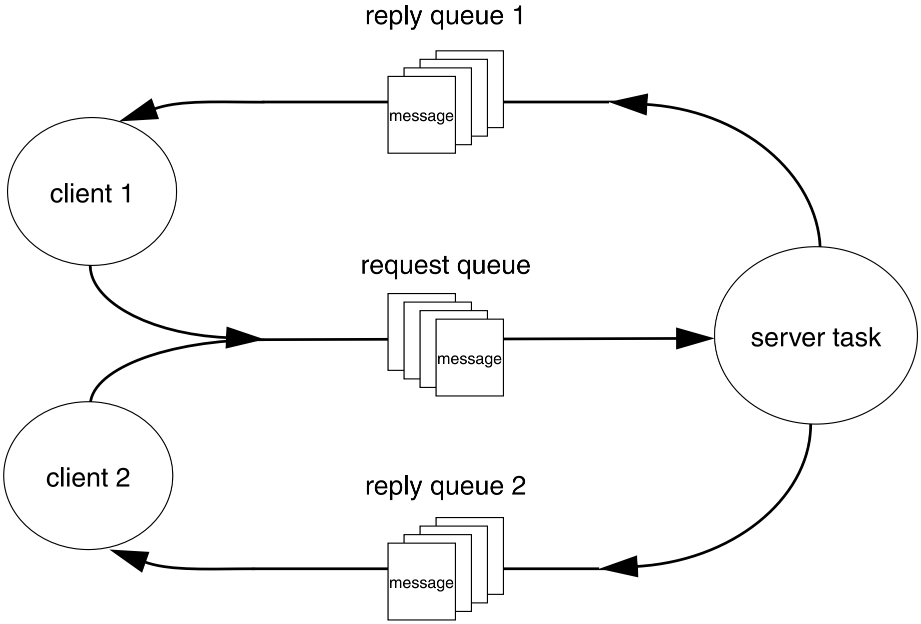
Servers and Clients with Message Queues

Real-time systems are often structured using a *client-server* model of tasks. In this model, server tasks accept requests from client tasks to perform some service, and usually return a reply. The requests and replies are usually made in the form of intertask messages. In VxWorks, message queues or pipes (see *2.3.5 Pipes*, p.53) are a natural way to implement this.

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

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.

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



Message Queues and VxWorks Events

This section describes using VxWorks events with message queues. You can also use VxWorks events with other VxWorks objects. For more information, see *2.4 VxWorks Events*, p.57.

Using Events

A message queue can send events to a task, if it is requested to do so by the task. To request that a message queue send events, a task must register with the message queue using **msgQEvStart( )**. From that point on, every time the message queue receives a message and there are no tasks pending on it, the message queue sends events to the registered task. To request that the message queue stop sending events, the registered task calls **msgQEvStop( )**.

Only one task can be registered with a message queue at any given time. The events a message queue sends to a task can be retrieved by the task using routines in **eventLib**. Details on when message queues send events are documented in the reference entry for **msgQEvStart( )**.

In some applications, the creator of a message queue may wish to know when a message queue failed to send events. Such a scenario can occur if a task registers with a message queue, and is subsequently deleted before having time to unregister. In this situation, a send operation could cause the message queue to attempt to send events to the deleted task. Such an attempt would obviously fail. If the message queue is created with the **SG_Q_EVENTSEND_ERROR_NOTIFY** option, the send operation returns an error. Otherwise, VxWorks handles the error quietly.

Using **eventReceive( )**, a task may pend on events meant to be sent by a message queue. If the message queue is deleted, the task pending on events is returned to the ready state, just like the tasks that may be pending on the message queue itself.

Existing VxWorks API

The VxWorks events implementation does not propose to keep track of all the resources a task is currently registered with. Therefore, a resource can attempt to send events to a task that no longer exists. For example, a task may be deleted or may self-destruct while still registered with a resource to receive events. This error is detected only when the resource becomes free, and is reported by having **msgQSend( )** return **ERROR**. However, in this case the error does not mean the semaphore was not given or that the message was not properly delivered. It simply means the resource could not send events to the registered task. This is a different behavior than the one presently in place under VxWorks, however it is the same behavior that exists for pSOS message queues.

Performance Impact

There is no performance impact on **msgQSend( )** when a task is pending for the message queue. However, when this is not the case, the call to **msgQSend( )** takes longer to complete, since events may have to be sent to a task. Furthermore, the call may unpend a task waiting for events, which means the caller may be preempted, even if no task is waiting for the message.

The **msgQDestroy( )** routine performance is impacted in cases where a task is waiting for events from the message queue, since the task has to be awakened. Also note that, in this case, events need not be sent.

2.3.5 Pipes

Pipes provide an alternative interface to the message queue facility that goes through the VxWorks I/O system. Pipes are virtual I/O devices managed by the driver **pipeDrv**. The routine **pipeDevCreate( )** creates a pipe device and the underlying message queue associated with that pipe. The call specifies the name of the created pipe, the maximum number of messages that can be queued to it, and the maximum length of each message:

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

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

As I/O devices, pipes provide one important feature that message queues cannot—the ability to be used with **select( )**. This routine allows a task to wait for data to be available on any of a set of I/O devices. The **select( )** routine also works with other asynchronous I/O devices including network sockets and serial devices. Thus, by using **select( )**, a task can wait for data on a combination of several pipes, sockets, and serial devices; see *4.3.8 Pending on Multiple File Descriptors: The Select Facility*, p. 117.

Pipes allow you to implement a client-server model of intertask communications; see *Servers and Clients with Message Queues*, p. 50.

2.3.6 Network Intertask Communication

Sockets

In VxWorks, the basis of intertask communications across the network is *sockets*. A socket is an endpoint for communications between tasks; data is sent from one socket to another. When you create a socket, you specify the Internet communications protocol that is to transmit the data. VxWorks supports the Internet protocols TCP and UDP. VxWorks socket facilities are source compatible with BSD 4.4 UNIX.

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

UDP provides a simpler but less robust form of communication. In UDP communications, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams.* A process creates a datagram socket and binds it to a particular port. There is no notion of a UDP "connection." Any UDP socket, on any host in the network, can send messages to any other UDP socket by specifying its Internet address and port number.

One of the biggest advantages of socket communications is that it is "homogeneous." Socket communications among processes are exactly the same regardless of the location of the processes in the network, or the operating system under which they are running. Processes can communicate within a single CPU, across a backplane, across an Ethernet, or across any connected combination of networks. Socket communications can occur between VxWorks tasks and host system processes in any combination. In all cases, the communications look identical to the application, except, of course, for their speed.

For more information, see *VxWorks Network Programmer's Guide: Networking APIs* and the reference entry for **sockLib**.

Remote Procedure Calls (RPC)

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

As discussed in the previous sections on message queues and pipes, many real-time systems are structured with a client-server model of tasks. In this model, client tasks request services of server tasks, and then wait for their reply. RPC formalizes this model and provides a standard protocol for passing requests and returning replies. Also, RPC includes tools to help generate the client interface routines and the server skeleton.

For more information about RPC, see *VxWorks Network Programmer's Guide: RPC, Remote Procedure Calls*.

2.3.7 Signals

VxWorks supports a software signal facility. Signals asynchronously alter the control flow of a task. Any task or ISR can raise a signal for a particular task. The task being signaled immediately suspends its current thread of execution and executes the task-specified signal handler routine the next time it is scheduled to run. The signal handler executes in the receiving task's context and makes use of that task's stack. The signal handler is invoked even if the task is blocked.

Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism. In general, signal handlers should be treated like ISRs; no routine should be called from a signal handler that might cause the handler to block. Because signals are asynchronous, it is difficult to predict which resources might be unavailable when a particular signal is raised. To be perfectly safe, call only those routines that can safely be called from an ISR (see Table 2-21). Deviate from this practice only when you are sure your signal handler cannot create a deadlock situation.

The *wind* kernel supports two types of signal interface: UNIX BSD-style signals and POSIX-compatible signals. The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b. For more information, see *3.9 POSIX Queued Signals*, p.105. For the sake of simplicity, we recommend that you use only one interface type in a given application, rather than mixing routines from different interfaces.

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

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**.

Basic Signal Routines

By default, VxWorks uses the basic signal facility component **INCLUDE_SIGNALS**. This component automatically initializes signals with **sigInit( )**. Table 2-15 shows the basic signal routines.

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

Table 2-15 **Basic Signal Calls (BSD and POSIX 1003.1b)**

POSIX 1003.1b Compatible Call	UNIX BSD Compatible Call	Description
signal( )	**signal( )**	Specifies the handler associated with a signal.
kill( )	**kill( )**	Sends a signal to a task.
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( )
	sigsetmask( )	Manipulates a signal mask.

sigprocmask( )	**sigsetmask( )**	Sets the mask of blocked signals.
sigprocmask( )	**sigblock( )**	Adds to a set of blocked signals.

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

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

Signal Configuration

The basic signal facility is included in VxWorks by default with **INCLUDE_SIGNALS** component.

2.4 VxWorks Events

VxWorks events introduce functionality similar to pSOS events into VxWorks 5.5. VxWorks events are included in the standard VxWorks facilities and are used to port pSOS events functionality to VxWorks. This section provides a brief summary of VxWorks events; then it describes pSOS events and VxWorks events in detail, comparing them and their APIs.

NOTE: This section uses the term *events* to describe pSOS and VxWorks events. These references are not to be confused with WindView events.

VxWorks events are a means of communication between tasks and interrupt routines (ISRs), between tasks and other tasks, or between tasks and VxWorks objects. In the context of VxWorks events, these objects are referred to as *resources*, and they include semaphores and message queues. Only tasks can receive events; whereas tasks, ISRs, or resources can send them.

In order for a task to receive events from a resource, the task must register with the resource. In order for the resource to send events, the resource must be free. The communication between tasks and resources is peer-to-peer, meaning that only the registered task can receive events from the resource. In this respect, events are like signals, in that they are directed at one task. A task, however, can wait on events from multiple resources; thus, it can be waiting for a semaphore to become free and for a message to arrive in a message queue.

Events are synchronous in nature (unlike signals), meaning that a receiving task must block or pend while waiting for the events to occur. When the desired events are received, the pending task continues its execution, as it would after a call to **msgQReceive( )** or **semTake( )**, for example. Thus, unlike signals, events do not require a handler.

Tasks can also wait on events that are not linked to resources. These are events that are sent from another task or from an ISR. A task does not register to receive these events; the sending task or ISR simply has to know of the task's interest in receiving the events. As an example, this scenario is similar to having an ISR give a binary semaphore, knowing there is a task interested in obtaining that semaphore.

The meaning of each event differs for each task. For example, when an event, **eventX**, is received, it can be interpreted differently by each task that receives it. Also, once an event is received by a task, the event is ignored if it is sent again to the same task. Consequently, it is not possible to track the number of times each event has been sent to a task.

WARNING: Because events are not, and cannot be, reserved, two independent applications can attempt to use the same events on the same task. As a precaution, middleware applications using VxWorks events should always publish a list of the events they are using.

2.4.1 pSOS Events

This section describes the functionality of pSOS events. This functionality provides the basis of VxWorks events, but does not fully describe their behavior. For details, see *VxWorks Enhancements to pSOS Events*, p.61.

Sending and Receiving Events

In the pSOS operating system, events can be sent from a resource to a task, from an ISR to a task, or directly between two tasks. Tasks, ISRs, and resources all use the same **ev_send( )** API to send events.

For a task to receive events from a resource, the task must register with that resource and request it to send a specific set of events when it becomes free. The resource is either a semaphore or a message queue. When the resource becomes free, it sends the set of events to the registered task. This task may, or may not be, waiting for the events.

As mentioned above, a task can also receive events from another task. For example, if two tasks agree to send events between them, **taskA** could send **taskB** a specific events set when it (**taskA**) finishes executing, to let **taskB** know that this has occurred. As with events sent from a resource, the receiving task may, or may not be, waiting for the events.

Waiting for Events

A task can wait for multiple events from one or more sources. Each source can send multiple events, and a task can also wait to receive only one event, or all events. For example, a task may be waiting for events 1 to 10, where 1 to 4 come from a semaphore, 5 to 6 come from a message queue, and 7 to 10 come from another task.

A task can also specify a timeout when waiting for events.

Registering for Events

Only one task can register itself to receive events from a resource. If another task subsequently registers with the same resource, the previously registered task is automatically unregistered, without its knowledge. This behavior differs in VxWorks events. For details, see *VxWorks Enhancements to pSOS Events*, p. 61.

When a task registers with a resource, events are not sent immediately, even if the resource is free at the time of registration. The events are sent the next time the resource becomes free. For example, a semaphore will send events the next time it is given, as long as no task is waiting for it. (Being given does not always mean that a resource is free; see *Freeing Resources*, p. 59). This behavior is configurable for VxWorks events. For details, see *VxWorks Enhancements to pSOS Events*, p. 61.

Freeing Resources

When a resource sends events to a task to indicate that it is free, it does not mean that resource is reserved. Therefore, a task waiting for events from a resource will unpend when the resource becomes free, however the resource may be taken in the meantime. There are no guarantees that the resource will still be available, if the task subsequently attempts to take ownership of it.

As mentioned above, a resource only sends events when it becomes free. This is not synonymous with *being released*. To clarify, if a semaphore is *given*, it is actually being released. However, it is not considered free if another task is waiting for it at the time it is released. Therefore, in cases where two or more tasks are constantly exchanging ownership of a resource, that resource never becomes free; thus, it may never send events.

pSOS Events API

The pSOS events API routines are listed in Table 2-16:

Table 2-16 **pSOS Events API**

Routine	Meaning
ev_send( )	Sends events to a task.
ev_receive( )	Waits for events.
sm_notify( )	Registers a task to be notified of semaphore availability.

Table 2-16 **pSOS Events API**

Routine	Meaning
q_notify( )	Registers a task to be notified of message arrival on a message queue.
q_vnotify( )	Registers a task to be notified of message arrival on a variable-length message queue.

2.4.2 VxWorks Events

The implementation of VxWorks events is based on the way pSOS events work. This section first clarifies some of the crucial terms used to discuss VxWorks events. Then it describes VxWorks events in more detail, comparing their functionality to that of pSOS events.

Free Resource Definition

A key concept in understanding events sent by resources, is that resources send events when they become free. Thus, it is crucial to define what it means for a resource to be *free* for VxWorks events.

Mutex Semaphore

A mutex semaphore is considered free when it no longer has an owner and no one is pending on it. For example, following a call to **semGive( )**, the semaphore will not send events if another task is pending on a **semTake( )** for the same semaphore.

Binary Semaphore

A binary semaphore is considered free when no task owns it and no one is waiting for it.

Counting Semaphore

A counting semaphore is considered free when its count is nonzero and no one is pending on it. Thus, events cannot 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 one is pending for the arrival of a message in that queue. Thus, events cannot be used as a mechanism to compute the number of messages sent to a message queue.

VxWorks Enhancements to pSOS Events

When VxWorks events were implemented, some enhancements were made to the basic pSOS functionality. This section describes those enhancements and configuration options, and compares the resulting behavior of VxWorks events with pSOS events.

- **SIngle Task Resource Registration.** As mentioned in *2.4.1 pSOS Events*, p.58, under pSOS, when a task registers with a resource to send pSOS events, it can inadvertently deregister another task that had previously registered with the resource. This prevents the first task from receiving events from the resource with which it registered. Consequently, the task that first registered with the resource could stay in a pend state indefinitely.

  In order to solve this problem, VxWorks events provide an option whereby the second task is not allowed to register with the resource, if another task is already registered with it. If a second task tries to register with the resource, an error is returned. In VxWorks, you can configure the registration mechanism to use either the VxWorks or the pSOS behavior.

- **Option for Immediate Send.** As mentioned in *Registering for Events*, p.59, when a pSOS task registers with a resource, the resource does not send events to the task immediately, even if it is free at the time of registration. For VxWorks events, the default behavior is the same. However, VxWorks events provide an option that allows a task, at the time of registration, to request that the resource send the events immediately, if the resource is free at the time of registration.

- **Option for Automatic Unregister.** There are situations in which a task may want to receive events from a resource only once, and then unregister. The pSOS implementation requires a task to explicitly unregister after having received events from the resource. The VxWorks implementation provides an option whereby a registering task can tell the resource to only send events once, and automatically unregister the task when this occurs.

- **Automatic Unpend upon Resource Deletion.** When a resource (a semaphore or message queue) is deleted, the **semDelete( )** and **msgQDelete( )** implementation unpends any task. This prevents the task from pending indefinitely, while waiting for events from the resource being deleted. The pending task then resumes execution, and receives an **ERROR** return value from the **eventReceive( )** call that caused the task to pend. See also, *Existing VxWorks API*, p.46 and *Existing VxWorks API*, p.52.

Task Events Register

Each task has its own events field or container, referred to as the *task events register*. The task events register is a per task 32-bit field used to store the events that a task receives from resources, ISRs, and other tasks.

You do not access the task events register directly. Tasks, ISRs, and resources fill the events register of a particular task by sending events to that task. A task can also send itself events, thereby filling its own events register. Events 25 to 32 (VXEV25 or 0x01000000 to VXEV32 or 0x80000000) are reserved for system use only, and are not available to VxWorks users. Table 2-17 describes the routines that affect the contents of the events register.

Table 2-17 **Event Register Routines**

Routine	Effects
eventReceive( )	Clears or leaves the contents of the event register intact, depending on the options selected.
eventClear( )	Clears the contents of the event register
eventSend( )	Copies events into the event register.
semGive( )	Copies events into the event register, if a task is registered with the semaphore.
msgQSend( )	Copies events into the event register, if a task is registered with the message queue.

VxWorks Events API

For details on the API for VxWorks events, see the reference entries for **eventLib**, **semEvLib**, and **msgQEvLib**.

Show Routines

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( )**

2.4.3 API Comparison

The VxWorks events API has made modifications to the pSOS events API for the purpose of better describing the action of the routines. The pSOS API uses a family of *notify* routines for registering and unregistering from a resource. However, these names do not correctly reflect the action of resources, which is to either *send*, or *not send*, events. Thus, the VxWorks API more precisely describes the request for a resource to start sending events and to stop sending events to a task. This implementation uses the **semEvStart( )** and **msgQEvStart( )** routines to tell resources to start sending events, and the **semEvStop( )** and **msgQEvStop( )** routines to tell a resource to stop sending events.

Table 2-18 compares the similarities and differences between the VxWorks and pSOS events API:

Table 2-18 **Comparison of Events**

VxWorks Routine	pSOS Routine	Comments
eventSend	**ev_send**	Direct port
eventReceive	**ev_receive**	Direct port
eventClear		New functionality in VxWorks.
semEvStart	**sm_notify**	**semEvStart** is equivalent to calling **sm_notify** with a nonzero events argument.
semEvStop	**sm_notify**	**semEvStop** is equivalent to calling **sm_notify** with an events argument equal to 0.
msgQEvStart	**q_vnotify**	**msgQEvStart** is equivalent to calling **q_notify** with a nonzero events argument.

Table 2-18 **Comparison of Events**

VxWorks Routine	pSOS Routine	Comments
msgQEvStop	**q_vnotify**	**msgQEvStop** is equivalent to calling **q_notify** with an events argument equal to 0.
	q_notify	VxWorks does not have a fixed-length message queue mechanism.

2.5 Watchdog Timers

VxWorks includes a watchdog-timer mechanism that allows any C function to be connected to a specified time delay. Watchdog timers are maintained as part of the system clock ISR. For information about POSIX timers, see *3.2 POSIX Clocks and Timers*, p.73.

Functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. However, if the kernel is unable to execute the function immediately for any reason (such as a previous interrupt or kernel state), the function is placed on the **tExcTask** work queue. Functions on the **tExcTask** work queue execute at the priority level of the **tExcTask** (usually 0).

Restrictions on ISRs apply to routines connected to watchdog timers. The functions in Table 2-19 are provided by the **wdLib** library.

Table 2-19 **Watchdog Timer Calls**

Call	Description
wdCreate( )	Allocates and initializes a watchdog timer.
wdDelete( )	Terminates and deallocates a watchdog timer.
wdStart( )	Starts a watchdog timer.
wdCancel( )	Cancels a currently counting watchdog timer.

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

Example 2-4 **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);
    /* ... */
    }
```

2.6 Interrupt Service Code: 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. For the fastest possible response to interrupts, VxWorks runs interrupt service routines (ISRs) in a special context outside of any task's context. Thus, interrupt handling involves no task context switch. Table 2-20 lists the interrupt routines provided in **intLib** and **intArchLib**.

For boards with an MMU, the optional product VxVMI provides write protection for the interrupt vector table; see *12. Virtual Memory Interface*.

Table 2-20 **Interrupt Routines**

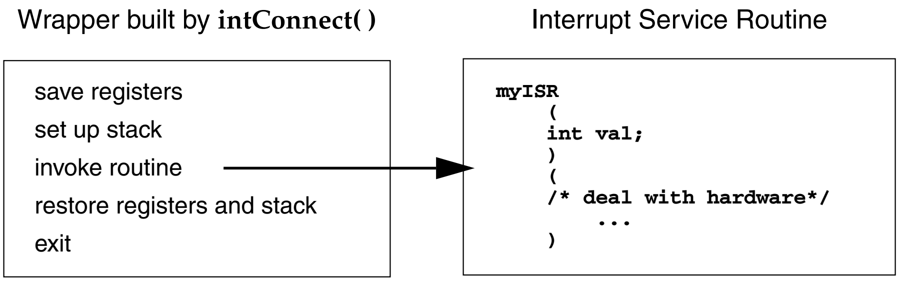
Call	Description
intConnect( )	Connects a C routine to an interrupt vector.
intContext( )	Returns TRUE if called from interrupt level.
intCount( )	Gets the current interrupt nesting depth.
intLevelSet( )	Sets the processor interrupt mask level.
intLock( )	Disables interrupts.
intUnlock( )	Re-enables interrupts.
intVecBaseSet( )	Sets the vector base address.
intVecBaseGet( )	Gets the vector base address.
intVecSet( )	Sets an exception vector.
intVecGet( )	Gets an exception vector.

2.6.1 Connecting Routines to Interrupts

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

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

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


Wrapper built by **intConnect( )**
Interrupt Service Routine
save registers
set up stack
invoke routine
restore registers and stack
exit

```
myISR
    (
    int val;
    )
    (
    /* deal with hardware*/
        ...
    )
```

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

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

2.6.2 Interrupt Stack

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

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

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

2.6.3 Writing and Debugging ISRs

There are some restrictions on the routines you can call from an ISR. For example, you cannot use routines like **printf( )**, **malloc( )**, and **semTake( )** in your ISR. You can, however, use **semGive( )**, **logMsg( )**, **msgQSend( )**, and **bcopy( )**. For more information, see *2.6.4 Special Limitations of ISRs*, p.68.

2.6.4 Special Limitations of ISRs

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

Table 2-21 **Routines that Can Be Called by Interrupt Service Routines**

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

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

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

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

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

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

All VxWorks utility libraries, such as the linked-list and ring-buffer libraries, can be used by ISRs. As discussed earlier (*2.2.6 Task Error Status: errno*, p.22), the global variable **errno** is saved and restored as a part of the interrupt enter and exit code generated by the **intConnect( )** facility. Thus, **errno** can be referenced and modified by ISRs as in any other code. Table 2-21 lists routines that can be called from ISRs.

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

One example of such an exception is the following message:

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

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

2.6.6 Reserving High Interrupt Levels

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

⚠ **CAUTION:** Some hardware prevents masking certain interrupt levels; check the hardware manufacturer's documentation.

2.6.7 Additional Restrictions for ISRs at High Interrupt Levels

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

- The ISR can be connected only with **intVecSet( )**.
- The ISR cannot use any VxWorks operating system facilities that depend on interrupt locks for correct operation. The effective result is that the ISR cannot safely make any call to any VxWorks function, except reboot.

For more information, see the VxWorks architecture supplement document for the architecture in question.

WARNING: The use of NMI with any VxWorks functionality, other than reboot, is not recommended. Routines marked as “interrupt safe” do not imply they are NMI safe and, in fact, are usually the very ones that NMI routines must not call (because they typically use **intLock( )** to achieve the interrupt safe condition).

2.6.8 Interrupt-to-Task Communication

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

- **Shared Memory and Ring Buffers.** ISRs can share variables, buffers, and ring buffers with task-level code.
- **Semaphores.** ISRs can give semaphores (except for mutual-exclusion semaphores and VxMP shared semaphores) that tasks can take and wait for.
- **Message Queues.** ISRs can send messages to message queues for tasks to receive (except for shared message queues using VxMP). If the queue is full, the message is discarded.
- **Pipes.** ISRs can write messages to pipes that tasks can read. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message written is discarded because the ISR cannot block. ISRs must not invoke any I/O routine on pipes other than **write( )**.
- **Signals.** ISRs can “signal” tasks, causing asynchronous scheduling of their signal handlers.

3
POSIX Standard Interfaces

3.1 Introduction

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

This chapter uses the qualifier "Wind" to identify facilities designed expressly for use with the VxWorks *wind* kernel. For example, you can find a discussion of Wind semaphores contrasted to POSIX semaphores in *3.6.1 Comparison of POSIX and Wind Semaphores*, p.86.

POSIX asynchronous Input/Output (AIO) routines are available in the **aioPxLib** library. The VxWorks AIO implementation meets the specification in the POSIX 1003.1b standard. For more information, see *4.6 Asynchronous Input/Output*, p.123.

3.2 POSIX Clocks and Timers

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

The POSIX standard provides a means of identifying multiple virtual clocks, but only one clock is required—the system-wide real-time clock. No virtual clocks are supported in VxWorks.

The system-wide real-time clock is identified in the clock and timer routines as **CLOCK_REALTIME**, and is defined in **time.h**. VxWorks provides routines to access the system-wide real-time clock. For more information, see the reference entry for **clockLib**.

The POSIX timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to create, set, and delete a timer. For more information, see the reference entry for **timerLib**. When a timer goes off, the default signal, **SIGALRM**, is sent to the task. To install a signal handler that executes when the timer expires, use the **sigaction( )** routine (see *2.3.7 Signals*, p. 55).

Example 3-1 **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);
    }
```

An additional POSIX function, **nanosleep( )**, provides specification of sleep or delay time in units of seconds and nanoseconds, in contrast to the ticks used by the Wind **taskDelay( )** function. Nevertheless, the precision of both is the same, and is determined by the system clock rate. Only the units differ.

3.3 POSIX Memory-Locking Interface

Many operating systems perform memory *paging* and *swapping*, which copy blocks of memory out to disk and back. These techniques allow you to use more virtual memory than there is physical memory on a system. However, because they impose severe and unpredictable delays in execution time, paging and swapping are undesirable in real-time systems. Consequently, the *wind* kernel never uses them.

However, the POSIX 1003.1b standard for real-time extensions is also used with operating systems that do 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.

To facilitate porting programs between other POSIX-conforming systems and VxWorks, VxWorks includes the POSIX page-locking routines. The routines have no adverse affect in VxWorks systems, because all memory is essentially always locked.

The POSIX page-locking routines are part of the memory management library, **mmanPxLib**, and are listed in Table 3-1. When used in VxWorks, these routines do nothing except return a value of **OK** (0), since all pages are always kept in memory.

Table 3-1 **POSIX Memory Management Calls**

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

To include the **mmanPxLib** library, configure VxWorks with the **INCLUDE_POSIX_MEM** component.

3.4 POSIX Threads

POSIX threads are similar to tasks, but with some additional characteristics, including a thread ID that differs from its task ID.

3.4.1 POSIX Thread Attributes

POSIX characteristics are called *attributes*. Each attribute contains a set of values, and a set of *access functions* to retrieve and set those values. You can specify all thread attributes in an attributes object, **pthread_attr_t**, at thread creation. In a few cases, you can dynamically modify the attribute values in a running thread.

The POSIX attributes and their corresponding access functions are described below.

Stack Size

The **stacksize** attribute specifies the size of the stack to be used. This value can be rounded up to a page boundary.

- Attribute Name: **stacksize**
- Default Value: Uses the default stack size set for **taskLib**.
- Access Functions: **pthread_attr_getstacksize( )** and **pthread_attr_setstacksize( )**

Stack Address

The **stackaddr** attribute specifies the base of a region of user allocated memory to be used as a stack region for the created thread. Because the default value is **NULL**, the system should allocate a stack for the thread when it is created.

- Attribute Name: **stackaddr**
- Default Value: **NULL**
- Access Functions: **pthread_attr_getstackaddr( )** and **pthread_attr_setstackaddr( )**

Detach State

The **detachstate** attribute describes the state of a thread. With POSIX threads, the creator of a thread can block until the thread exits (see the entries for **pthread_exit( )** and **pthread_join( )** in the *VxWorks API Reference*). In this case, the thread is a **joinable** thread. Otherwise, it is a **detached** thread. A thread that was created as a **joinable** thread can dynamically make itself a **detached** thread by calling **pthread_detach( )**.

- Attribute Name: **detachstate**
- Possible Values: **PTHREAD_CREATE_DETACHED** and **PTHREAD_CREATE_JOINABLE**
- Default Value: **PTHREAD_CREATE_JOINABLE**
- Access Functions: **pthread_attr_getdetachstate( )** and **pthread_attr_setdetachstate( )**
- Dynamic Access Function: **pthread_detach( )**

Contention Scope

The **contentionscope** attribute describes how threads compete for resources, namely the CPU. Under VxWorks, all tasks compete for the CPU, so the competition is system-wide. Although POSIX allows two values, only **PTHREAD_SCOPE_SYSTEM** is implemented for VxWorks.

- Attribute Name: **contentionscope**
- Possible Values: **PTHREAD_SCOPE_SYSTEM** only (**PTHREAD_SCOPE_PROCESS** not implemented for VxWorks)
- Default Value: **PTHREAD_SCOPE_SYSTEM**
- Access Functions: **pthread_attr_getscope( )** and **pthread_attr_setscope( )**

Inherit Scheduling

The **inheritsched** attribute determines whether the thread is created with scheduling parameters inherited from its parent thread, or with parameters that are explicitly specified.

- Attribute Name: **inheritsched**
- Possible Values: **PTHREAD_EXPLICIT_SCHED** or **PTHREAD_INHERIT_SCHED**
- Default Value: **PTHREAD_INHERIT_SCHED**
- Access Functions: **pthread_attr_getinheritsched( )** and **pthread_attr_setinheritsched( )**

Scheduling Policy

The **schedpolicy** attribute describes the scheduling policy for the thread, and is valid only if the value of the **inheritsched** attribute is **PTHREAD_EXPLICIT_SCHED**.

- Attribute Name: **schedpolicy**
- Possible Values: **SCHED_FIFO** (preemptive priority scheduling) and **SCHED_RR** (round-robin scheduling by priority)
- Default Value: **SCHED_RR**
- Access Functions:**pthread_attr_getschedpolicy*( )*** and **pthread_attr_setschedpolicy( )**

Note that because the default value for the **inheritsched** attribute is **PTHREAD_INHERIT_SCHED**, the **schedpolicy** attribute is not used by default. For more information, see *3.5.3 Getting and Displaying the Current Scheduling Policy,* p.84.

Scheduling Parameters

The **schedparam** attribute describes the scheduling parameters for the thread, and is valid only if the value of the **inheritsched** attribute is **PTHREAD_EXPLICIT_SCHED**.

- Attribute Name: **schedparam**
- Range of Values: 0-255
- Default Value: Uses default task priority set for **taskLib**.
- Access Functions: **pthread_attr_getschedparam( )** and **pthread_attr_setschedparam( )**
- Dynamic Access Functions: **pthread_getschedparam( )** and **pthread_setschedparam( )** using thread ID, or **sched_getparam( )** and **sched_setparam( )** using task ID

Note that because the default value the **inheritsched** attribute is **PTHREAD_INHERIT_SCHED**, the **schedparam** attribute is not used by default. For more information, see *3.5.2 Getting and Setting POSIX Task Priorities*, p.82.

Specifying Attributes when Creating pThreads

Following are examples of creating a thread using the default attributes and using explicit attributes.

Example 3-2 **Creating a pThread Using Explicit Scheduing 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 3-3 **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 3-4 **Designating Your Own Stack for a pThread**

```
pthread_attr_init(&attr);

/* allocate memory for a stack region for the thread */
stackbase = malloc(2 * 4096);

if (stackbase == NULL)
    {
    printf("FAILED: mystack: malloc failed\n");
    exit(-1);
    }

/* set the stack pointer to the base address */
stackptr = (void *)((int)stackbase);

/* explicitly set the stackaddr attribute */
pthread_attr_setstackaddr(&attr, stackptr);

/* set the stacksize attribute to 4096 */
pthread_attr_setstacksize(&attr, (4096));
```

```
/* set the schedpolicy attribute to SCHED_FIFO */
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

/* create the pthread */
ret = pthread_create(&tid, &attr, mystack_thread, 0);
```

3.4.2 Thread Private Data

When a thread needs access to private data, POSIX uses a *key* to access that data. A location is created by calling to **pthread_key_create( )** and released by calling **pthread_key_delete( )**. The location is then accessed by calling **pthread_getspecific( )** and **pthread_setspecific( )**. The **pthread_key_create( )** routine has an option for a *destructor function*, which is called when the creating thread exits, if the value associated with the key is non-NULL.

3.4.3 Thread Cancellation

POSIX provides a mechanism, called *cancellation*, to terminate a thread gracefully. There are two types of cancellation, *synchronous* and *asynchronous*. Synchronous cancellation causes the thread to explicitly check to see if it was cancelled or to call a function that contains a *cancellation point*. Asynchronous cancellation causes the execution of the thread to be interrupted and a handler to be called, much like a signal.[1]

Routines that can be used with cancellation are listed in Table 3-2.

Table 3-2 **Thread Cancellation Routines**

Routine	Meaning
pthread_setcancelstate( )	Enables or disables cancellation.
pthread_setcanceltype( )	Selects synchronous or asynchronous cancellation.
pthread_cleanup_push( )	Registers a function to be called when the thread is cancelled.
pthread_cleanup_pop( )	Unregisters a function to be called when a thread is cancelled, and then calls the function.

[1] Asynchronous cancellation is actually implemented with a special signal, **SIGCANCEL**, which users should be careful to not block or ignore.

A thread can register and unregister functions to be called when it is cancelled by **pthread_cleanup_push( )** and **pthread_cleanup_pop( )**. The **pthread_cleanup_pop( )** routine can optionally call the function when unregistering it.

3.5 POSIX Scheduling Interface

The POSIX 1003.1b scheduling routines, provided by **schedPxLib**, are shown in Table 3-3. These routines let you use a portable interface to get and set task priority, get the scheduling policy, get the maximum and minimum priority for tasks, and if round-robin scheduling is in effect, get the length of a time slice. This section describes how to use these routines, beginning with a list of the minor differences between the POSIX and Wind methods of scheduling.

Table 3-3 **POSIX Scheduling Calls**

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

To include the **schedPxLib** library of POSIX scheduling routines, configure VxWorks with the **INCLUDE_POSIX_SCHED** component.

3.5.1 Comparison of POSIX and Wind Scheduling

POSIX and Wind scheduling routines differ in the following ways:

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

3.5.2 Getting and Setting POSIX Task Priorities

The routines **sched_setparam( )** and **sched_getparam( )** set and get a task's priority, respectively. Both routines take a task ID and a **sched_param** structure (defined in *installDir***/target/h/sched.h**). A task ID of 0 sets or gets the priority for the calling task.

When **sched_setparam( )** is called, the **sched_priority** member of the **sched_param** structure specifies the new task priority. The **sched_getparam( )** routine fills in the **sched_priority** with the specified task's current priority.

Example 3-5 **Getting and Setting POSIX Task Priorities**

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

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

/* defines */
```

```
#define PX_NEW_PRIORITY 150

STATUS priorityTest (void)
    {
    struct sched_param myParam;

    /* initialize param structure to desired priority */

    myParam.sched_priority = PX_NEW_PRIORITY;
    if (sched_setparam (0, &myParam) == ERROR)
        {
        printf ("error setting priority\n");
        return (ERROR);
        }

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

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

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

    return (OK);
    }
```

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

The only way to change the scheduling policy is to change it for all tasks; there is no POSIX routine for this purpose. To set a system-wide scheduling policy, use the Wind function **kernelTimeSlice( )** described in *Round-Robin Scheduling*, p.12.

3.5.3 Getting and Displaying the Current Scheduling Policy

The POSIX routine **sched_getscheduler( )** returns the current scheduling policy. There are two valid scheduling policies in VxWorks: preemptive priority scheduling (in POSIX terms, **SCHED_FIFO**) and round-robin scheduling by priority (**SCHED_RR**). For more information, see *Scheduling Policy*, p.78.

Example 3-6 **Getting POSIX Scheduling Policy**

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

/* includes */

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

STATUS schedulerTest (void)
    {
    int policy;

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

    /* sched_getscheduler returns either SCHED_FIFO or SCHED_RR */

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

    return (OK);
    }
```

3.5.4 Getting Scheduling Parameters: Priority Limits and Time Slice

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

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

Example 3-7 **Getting the POSIX Round-Robin Time Slice**

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

/* includes */

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

STATUS rrgetintervalTest (void)
    {
    struct timespec slice;

    /* turn on round robin */

    kernelTimeSlice (30);

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

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

3.6 POSIX Semaphores

POSIX defines both *named* and *unnamed* semaphores, which have the same properties, but use slightly different interfaces. The POSIX semaphore library provides routines for creating, opening, and destroying both named and unnamed semaphores. When opening a named semaphore, you assign a symbolic name,[2] which the other named-semaphore routines accept as an argument. The POSIX semaphore routines provided by **semPxLib** are shown in Table 3-4.

[2] Some host operating systems, such as UNIX, require symbolic names for objects that are to be shared among processes. This is because processes do not normally share memory in such operating systems. In VxWorks, there is no requirement for named semaphores, because all kernel objects have unique identifiers. However, using named semaphores of the POSIX variety provides a convenient way of determining the object's ID.

Table 3-4 **POSIX Semaphore Routines**

Call	Description
semPxLibInit( )	Initializes the POSIX semaphore library (non-POSIX).
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.

To include the POSIX **semPxLib** library semaphore routines, configure VxWorks with the **INCLUDE_POSIX_SEM** component. The initialization routine **semPxLibInit( )** is called by default when POSIX semaphores have been included in VxWorks.

3.6.1 Comparison of POSIX and Wind Semaphores

POSIX semaphores are *counting* semaphores; that is, they keep track of the number of times they are given. The Wind semaphore mechanism is similar to that specified by POSIX, except that Wind semaphores offer additional features listed below. When these features are important, Wind semaphores are preferable.

- priority inheritance
- task-deletion safety
- the ability for a single task to take a semaphore multiple times
- ownership of mutual-exclusion semaphores
- semaphore timeouts
- the choice of queuing mechanism

The POSIX terms *wait* (or *lock*) and *post* (or *unlock*) correspond to the VxWorks terms *take* and *give*, respectively. The POSIX routines for locking, unlocking, and getting the value of semaphores are used for both named and unnamed semaphores.

The routines **sem_init( )** and **sem_destroy( )** are used for initializing and destroying unnamed semaphores only. The **sem_destroy( )** call terminates an unnamed semaphore and deallocates all associated memory.

The routines **sem_open( )**, **sem_unlink( )**, and **sem_close( )** are for opening and closing (destroying) named semaphores only. The combination of **sem_close( )** and **sem_unlink( )** has the same effect for named semaphores as **sem_destroy( )** does for unnamed semaphores. That is, it terminates the semaphore and deallocates the associated memory.

⚠ **WARNING:** When deleting semaphores, particularly mutual-exclusion semaphores, avoid deleting a semaphore still required by another task. Do not delete a semaphore unless the deleting task first succeeds in locking that semaphore. Similarly for named semaphores, close semaphores only from the same task that opens them.

3.6.2 Using Unnamed Semaphores

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

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

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

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

Example 3-8 **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);

void unnameSem (void)
    {
    sem_t * pSem;

    /* reserve memory for semaphore */
    pSem = (sem_t *) malloc (sizeof (sem_t));

    /* initialize semaphore to unavailable */
    if (sem_init (pSem, 0, 0) == -1)
        {
        printf ("unnameSem: sem_init failed\n");
        free ((char *) pSem);
        return;
        }

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

    /* do something useful to synchronize with syncTask */

    /* unlock sem */
    printf ("unnameSem: posting semaphore - synchronizing action\n");
    if (sem_post (pSem) == -1)
        {
        printf ("unnameSem: posting semaphore failed\n");
        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 */
    }
```

3.6.3 Using Named Semaphores

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

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

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

The results, based on the flags and whether the semaphore accessed already exists, are shown in Table 3-5. There is no entry for **O_EXCL** alone, because using that flag alone is not meaningful.

Table 3-5 **Possible Outcomes of Calling sem_open( )**

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

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

If VxWorks is configured with **INCLUDE_POSIX_SEM_SHOW**, you can use **show( )** from the shell to display information about a POSIX semaphore:[3]

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

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

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

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

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

Example 3-9 **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"
 */
```

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

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

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

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

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

    printf ("nameSem: spawning sync task\n");
    taskSpawn ("tSyncSemTask", 90, 0, 2000, syncSemTask, name);

    /* do something useful to synchronize with syncSemTask */

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

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

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

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

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

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

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

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

    /* do something useful here */

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

3.7 POSIX Mutexes and Condition Variables

Mutexes and condition variables provide compatibility with the POSIX standard (1003.1c). They perform essentially the same role as mutual exclusion and binary semaphores (and are in fact implemented using them). They are available with **pthreadLib.** 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**.

Protocol

The **protocol** mutex attribute describes how the mutex deals with the priority inversion problem described in the section for mutual-exclusion semaphores (*Mutual-Exclusion Semaphores*, p. 40).

- Attribute Name: **protocol**
- Possible Values: **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**
- Access Routines: **pthread_mutexattr_getprotocol( )** and **pthread_mutexattr_setprotocol( )**

To create a mutual-exclusion semaphore with *priority inheritance*, use the **SEM_Q_PRIORITY** and **SEM_PRIO_INHERIT** options to **semMCreate( )**. Mutual-exclusion semaphores created with the *priority protection* value use the notion of a *priority ceiling*, which is the other mutex attribute.

Priority Ceiling

The **prioceiling** attribute is the POSIX priority ceiling for a mutex created with the **protocol** attribute set to **PTHREAD_PRIO_PROTECT**.

- Attribute Name: **prioceiling**
- Possible Values: any valid (POSIX) priority value
- Access Routines: **pthread_mutexattr_getprioceiling( )** and **pthread_mutexattr_setprioceiling( )**
- Dynamic Access Routines: **pthread_mutex_getprioceiling( )** and **pthread_mutex_setprioceiling( )**

Note that the POSIX priority numbering scheme is the inverse of the Wind scheme. See *3.5.1 Comparison of POSIX and Wind Scheduling*, p. 82.

A priority ceiling is defined by the following conditions:

– Any thread attempting to acquire a mutex, whose priority is higher than the ceiling, cannot acquire the mutex.

– Any thread 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 thread's priority is restored to its previous value when the mutex is released.

3.8 POSIX Message Queues

The POSIX message queue routines, provided by **mqPxLib**, are shown in Table 3-6.

Table 3-6 **POSIX Message Queue Routines**

Call	Description
mqPxLibInit( )	Initializes the POSIX message queue library (non-POSIX).
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.

To configure VxWorks to include the POSIX message queue routine, include the **INCLUDE_POSIX_MQ** component. The initialization routine **mqPxLibInit( )** makes the POSIX message queue routines available, and is called automatically when the **INCLUDE_POSIX_MQ** component is included in the system.

3.8.1 Comparison of POSIX and Wind Message Queues

The POSIX message queues are similar to Wind message queues, except that POSIX message queues provide messages with a range of priorities. The differences between the POSIX and Wind message queues are summarized in Table 3-7.

Table 3-7 **Message Queue Feature Comparison**

Feature	Wind Message Queues	POSIX Message Queues
Message Priority Levels	1	32
Blocked Task Queues	FIFO or priority-based	Priority-based

Table 3-7 **Message Queue Feature Comparison**

Feature	Wind Message Queues	POSIX Message Queues
Receive with Timeout	Optional	Not available
Task Notification	Not available	Optional (one task)
Close/Unlink Semantics	No	Yes

POSIX message queues are also portable, if you are migrating to VxWorks from another 1003.1b-compliant system. This means that you can use POSIX message queues without having to change the code, thereby reducing the porting effort.

3.8.2 POSIX Message Queue Attributes

A POSIX message queue has the following attributes:

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

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

Example 3-10 **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);
}
```

3.8.3 Displaying Message Queue Attributes

The VxWorks **show( )** command produces a display of the key message queue attributes, for either POSIX or Wind message queues. To get information on POSIX message queues, configure VxWorks to include the **INCLUDE_POSIX_MQ_SHOW** component.

For example, if **mqPXId** is a POSIX message queue:

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

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

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

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

```
-> 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
```

NOTE: The built-in **show( )** routine handles Wind message queues; see the *Tornado User's Guide: Shell* for information on built-in routines. You can also use the Tornado browser to get information on Wind message queues; see the *Tornado User's Guide: Browser* for details.

3.8.4 Communicating Through a Message Queue

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

To put messages on a queue, use **mq_send( )**. If a task attempts to put a message on the queue when the queue is full, the task blocks until some other task reads a message from the queue, making space available. To avoid blocking on **mq_send( )**, set **O_NONBLOCK** when you open the message queue. In that case, when the queue is full, **mq_send( )** returns -1 and sets **errno** to **EAGAIN** instead of pending, allowing you to try again or take other action as appropriate.

One of the arguments to **mq_send( )** specifies a message priority. Priorities range from 0 (lowest priority) to 31 (highest priority); see *3.5.1 Comparison of POSIX and Wind Scheduling*, p.82.

When a task receives a message using **mq_receive( )**, the task receives the highest-priority message currently on the queue. Among multiple messages with the same priority, the first message placed on the queue is the first received (FIFO order). If the queue is empty, the task blocks until a message is placed on the queue.

To avoid pending on **mq_receive( )**, open the message queue with **O_NONBLOCK**; in that case, when a task attempts to read from an empty queue, **mq_receive( )** returns -1 and sets **errno** to **EAGAIN**.

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

Example 3-11 **POSIX Message Queues**

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

/* mqEx.h - message example header */
/* defines */
#define MQ_NAME "exampleMessageQueue"

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

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

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

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

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

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

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

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

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

/* sendTask.c - mq sending example */

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

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

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

    /* open msg queue; should already exist with default attributes */


    if ((mqPXId = mq_open (MQ_NAME, O_RDWR, 0, NULL)) == (mqd_t) -1)
        {
        printf ("sendTask: mq_open failed\n");
        return;
        }

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

3.8.5 Notifying a Task that a Message is Waiting

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

The **mq_notify( )** call specifies a signal to be sent to the task when a message is placed on an empty queue. This mechanism uses the POSIX data-carrying extension to signaling, which allows you, for example, to carry a queue identifier with the signal (see *3.9 POSIX Queued Signals*, p.105).

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

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

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

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

Example 3-12 **Notifying a Task that a Message Queue is Waiting**

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

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

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

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

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

int exMqNotify
    (
    char * pMess              /* text for message to self */
    )

    {
    struct mq_attr    attr;                  /* queue attribute structure */
    struct sigevent   sigNotify;             /* to attach notification */
    struct sigaction  mySigAction;           /* to attach signal handler */
    mqd_t             exMqId                 /* id of message queue */

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

    /*
     * Install signal handler for the notify signal 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  = O_NONBLOCK;              /* make nonblocking */
    attr.mq_maxmsg = 2;
    attr.mq_msgsize = MSG_SIZE;

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

```
/*
 * Set up notification: fill in a sigevent structure and pass it
 * to mq_notify(). The queue ID is passed as an argument to the
 * signal handler.
 */

sigNotify.sigev_signo      = SIGUSR1;
sigNotify.sigev_notify     = SIGEV_SIGNAL;
sigNotify.sigev_value.sival_int = (int) exMqId;

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

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

exMqRead (exMqId);

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

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

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

/* More cleanup */
if (mq_unlink (QNAM) == -1)
    {
    printf ("mq_unlink failed\n");
    return (-1);
    }
    return (0);
    }

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

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

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

    /*
     * Request notification again; it resets each time
     * a notification signal goes out.
     */

    sigNotify.sigev_signo = pInfo->si_signo;
    sigNotify.sigev_value = pInfo->si_value;
    sigNotify.sigev_notify = SIGEV_SIGNAL;

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

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

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

static void exMqRead
    (
    mqd_t      exMqId
    )
```

```
{
char        msg[MSG_SIZE];
int         prio;

/*
 * Read in the messages - uses a loop to read in the messages
 * because a notification is sent ONLY when a message is sent on
 * an EMPTY message queue. There could be multiple msgs if, for
 * example, a higher-priority task was sending them. Because the
 * message queue was opened with the O_NONBLOCK flag, eventually
 * this loop exits with errno set to EAGAIN (meaning we did an
 * mq_receive() on an empty message queue).
 */

while (mq_receive (exMqId, msg, MSG_SIZE, &prio) != -1)
    {
    printf ("exMqRead: received message: %s\n",msg);
    }

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

3.9 POSIX Queued Signals

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

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

VxWorks includes seven signals reserved for application use, numbered consecutively from **SIGRTMIN**. The presence of these reserved signals is required by POSIX 1003.1b, but the specific signal values are not; for portability, specify these signals as offsets from **SIGRTMIN** (for example, write **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 1003.1b also introduced an alternative means of receiving signals. The routine **sigwaitinfo( )** differs from **sigsuspend( )** or **pause( )** in that it allows your application to respond to a signal without going through the mechanism of a registered signal handler: when a signal is available, **sigwaitinfo( )** returns the value of that signal as a result, and does not invoke a signal handler even if one is registered. The routine **sigtimedwait( )** is similar, except that it can time out.

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

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

Call	Description
sigqueue( )	Sends a queued signal.
sigwaitinfo( )	Waits for a signal.
sigtimedwait( )	Waits for a signal with a timeout.

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

4
I/O System

4.1 Introduction

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

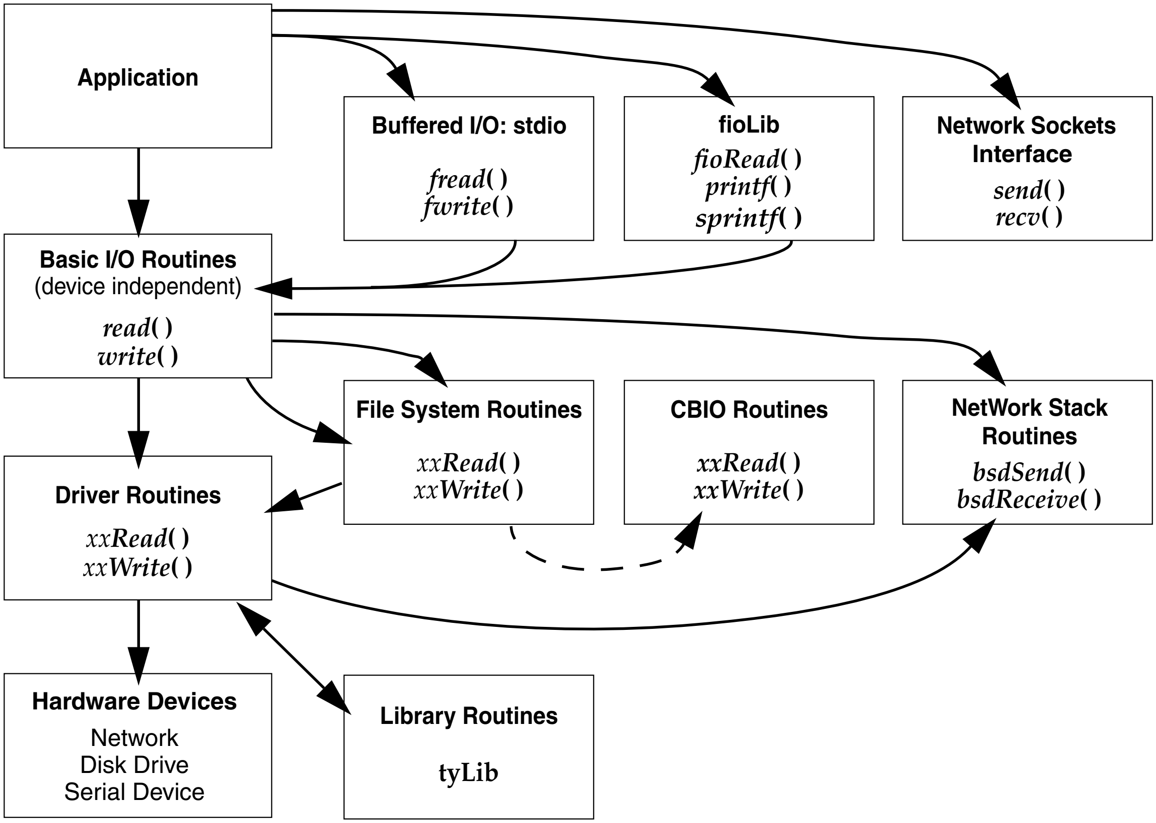
- character-oriented devices such as terminals or communications lines
- random-access block devices such as disks
- virtual devices such as intertask *pipes* and *sockets*
- monitor and control devices such as digital and analog I/O devices
- network devices that give access to remote devices

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

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

The diagram in Figure 4-1 illustrates the relationships between the different elements of the VxWorks I/O system. All of these elements are discussed in this chapter, except for file system routines (which are dealt with in *5. Local File Systems*), and the network elements (which are covered in the *VxWorks Network Programmer's Guide*).

Figure 4-1 **Overview of the VxWorks I/O System**



NOTE: In Figure 4-1, the dotted arrow line indicates that the CBIO block device is an optional path between File System Routines and Driver Routines.

4.2 Files, Devices, and Drivers

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

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

Consider the following named files:

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

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

Devices are handled by program modules called *drivers*. In general, using the I/O system does not require any further understanding of the implementation of devices and drivers. Note, however, that the VxWorks I/O system gives drivers considerable flexibility in the way they handle each specific device. Drivers conform to the conventional user view presented here, but can differ in the specifics. See *4.7 Devices in VxWorks*, p. 131.

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

4.2.1 Filenames and the Default Device

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

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

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

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

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

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

```
/usr
```

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

```
host:
```

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

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

```
DEV1:
```

NOTE: Filenames and directory names on dosFs devices are often separated by backslashes (****). These can be used interchangeably with forward slashes (**/**).

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

4.3 Basic I/O

Basic I/O is the lowest level of I/O in VxWorks. The basic I/O interface is source-compatible with the I/O primitives in the standard C library. There are seven basic I/O calls, shown in Table 4-1.

Table 4-1 **Basic I/O Routines**

Call	Description
creat( )	Creates a file.
delete( )	Deletes a file.
open( )	Opens a file. (Optionally, creates a file.)
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.

4.3.1 File Descriptors

At the basic I/O level, files are referred to by a *file descriptor*, or *fd*. An *fd* is a small integer returned by a call to **open( )** or **creat( )**. The other basic I/O calls take an *fd* as a parameter to specify the intended file.

A file descriptor is global to a system. For example, a task, **A**, that performs a **write( )** on *fd* 7 will write to the same file (and device) as a task, **B**, that performs a **write( )** on *fd* 7.

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

By default, file descriptors are reclaimed only when the file is closed.

4.3.2 Standard Input, Standard Output, and Standard Error

Three file descriptors are reserved and have special meanings:

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

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

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

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

Global Redirection

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

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

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

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

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

Task-Specific Redirection

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

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

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

4.3.3 Open and Close

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

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

The possible access flags are shown in Table 4-2.

Table 4-2 **File Access Flags**

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

⚠ **WARNING:** While the third parameter to **open( )** is usually optional for other operating systems, it is required for the VxWorks implementation of **open( )**. When the third parameter is not appropriate for any given call, it should be set to zero. Note that this can be an issue when porting software from UNIX to VxWorks.

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

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

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

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

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

```
close (fd);
```

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

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

4.3.4 Create and Delete

File-oriented devices must be able to create and delete files as well as open existing files.

The **creat( )** routine directs a file-oriented device to make a new file on the device and return a file descriptor for it. The arguments to **creat( )** are similar to those of **open( )** except that the filename specifies the name of the new file rather than an existing one; the **creat( )** routine returns an *fd* identifying the new file.

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

The **delete( )** routine deletes a named file on a file-oriented device:

```
delete ("name");
```

Do not delete files while they are open.

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

4.3.5 Read and Write

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

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

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

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

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

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

4.3.6 File Truncation

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

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

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

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

4.3.7 I/O Control

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

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

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

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

The discussion of specific devices in *4.7 Devices in VxWorks*, p.131 summarizes the **ioctl( )** functions available for each device. The **ioctl( )** control codes are defined in **ioLib.h**. For more information, see the reference entries for specific device drivers or file systems.

4.3.8 Pending on Multiple File Descriptors: The Select Facility

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

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

Tasks can block until data becomes available or the device is ready for writing. The **select( )** routine returns when one or more file descriptors are ready or a timeout has occurred. Using the **select( )** routine, a task specifies the file descriptors on which to wait for activity. Bit fields are used in the **select( )** call to specify the read and write file descriptors of interest. When **select( )** returns, the bit fields are modified to reflect the file descriptors that have become available. The macros for building and manipulating these bit fields are listed in Table 4-3.

Table 4-3 **Select Macros**

Macro	Function
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 1 if the specified bit is set; otherwise returns 0.

Applications can use **select( )** with any character I/O devices that provide support for this facility (for example, pipes, serial devices, and sockets). For information on writing a device driver that supports **select( )**, see *Implementing select( )*, p. 168.

Example 4-1 **The Select Facility**

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

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

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

/************************************************************************
* selServer - reads data as it becomes available from two different pipes
*
* Opens two pipe fds, reading from whichever becomes available. The
* server code assumes the pipes have been created from either another
* task or the shell. To test this code from the shell do the following:
*  -> ld < selServer.o
*  -> pipeDevCreate ("/pipe/highPriority", 5, 1024)
*  -> pipeDevCreate ("/pipe/normalPriority", 5, 1024)
*  -> fdHi = open   ("/pipe/highPriority", 1, 0)
*  -> fdNorm = open ("/pipe/normalPriority", 1, 0)
*  -> iosFdShow
*  -> sp selServer
*  -> i

* At this point you should see selServer's state as pended. You can now
* write to either pipe to make the selServer display your message.
*  -> write fdNorm, "Howdy", 6
*  -> write fdHi, "Urgent", 7
*/

STATUS selServer (void)
    {
    struct fd_set readFds;      /* bit mask of fds to read from */
    int      fds[MAX_FDS];      /* array of fds on which to pend */
    int      width;             /* number of fds on which to pend */
    int      i;                 /* index for fd array */
    char     buffer[MAX_DATA];  /* buffer for data that is read */
/* open file descriptors */
    if ((fds[0] = open (PIPEHI, O_RDONLY, 0)) == ERROR)
        {
        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);
            }
        }
    }
    }
```

4.4 Buffered I/O: stdio

The VxWorks I/O library provides a buffered I/O package that is compatible with the UNIX and Windows *stdio* package, and provides full ANSI C support. Configure VxWorks with the ANSI Standard component bundle to provide buffered I/O support.

NOTE: The implementation of **printf( )**, **sprintf( )**, and **sscanf( )**, traditionally considered part of the *stdio* package, is part of a different package in VxWorks. These routines are discussed in *4.5 Other Formatted I/O*, p. 122.

4.4.1 Using stdio

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

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

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

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

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

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

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

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

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

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

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

The **FILE** buffer is deallocated when **fclose( )** is called.

4.4.2 Standard Input, Standard Output, and Standard Error

As discussed in *4.3 Basic I/O*, p.111, 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 *fd*s. Each task using the standard I/O *fds* has its own *stdio* **FILE** buffers. The **FILE** buffers are deallocated when the task exits.

4.5 Other Formatted I/O

This section describes additional formatting routines and facilites.

4.5.1 Special Cases: printf( ), sprintf( ), and sscanf( )

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

Because these routines are implemented in this way, the full *stdio* package, which is optional, can be omitted from a VxWorks configuration without sacrificing their availability. Applications requiring *printf*-style output that is buffered can still accomplish this by calling **fprintf( )** explicitly to *stdout*.

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

4.5.2 Additional Routines: printErr( ) and fdprintf( )

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

4.5.3 Message Logging

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

4.6 Asynchronous Input/Output

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

The benefit of AIO is greater processing efficiency: it permits I/O operations to take place whenever resources are available, rather than making them await arbitrary events such as the completion of independent operations. AIO eliminates some of the unnecessary blocking of tasks that is caused by ordinary synchronous I/O; this decreases contention for resources between input/output and internal processing, and expedites throughput.

The VxWorks AIO implementation meets the specification in the POSIX 1003.1b standard. 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.

4.6.1 The POSIX AIO Routines

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

The default VxWorks initialization code calls **aioPxLibInit( )** automatically when the POSIX AIO component is included in VxWorks with **INCLUDE_POSIX_AIO**.

The **aioPxLibInit( )** routine takes one parameter, the maximum number of **lio_listio( )** calls that can be outstanding at one time. By default this parameter is **MAX_LIO_CALLS**. When the parameter is 0 (the default), the value is taken from **AIO_CLUST_MAX** (defined in *installDir***/target/h/private/aioPxLibP.h**).

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

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

Function	Description
aioPxLibInit( )	Initializes the AIO library (non-POSIX).
aioShow( )	Displays the outstanding AIO requests (non-POSIX).*
aio_read( )	Initiates an asynchronous read operation.
aio_write( )	Initiates an asynchronous write operation.
aio_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.

* This function is not built into the Tornado shell. To use it from the Tornado shell, VxWorks must be configured with the **INCLUDE_POSIX_AIO_SHOW** component. When you invoke the function, its output is sent to the standard output device.

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

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

Table 4-5 lists the names of the constants, and shows the constants used within initialization routines when the parameters are left at their default values of 0, and where these constants are defined.

Table 4-5 **AIO Initialization Functions and Related Constants**

Initialization Function	Configuration Parameter	Def. Value	Header File Constant used when arg = 0	Def. Value	Header File (all in *installDir*/target
aioPxLibInit( )	MAX_LIO_CALLS	0	AIO_CLUST_MAX	100	**h/private/aioPxLibP.h**
aioSysInit( )	MAX_AIO_SYS_TASKS	0	AIO_IO_TASKS_DFLT	2	**h/aioSysDrv.h**
	AIO_TASK_PRIORITY	0	AIO_IO_PRIO_DFLT	50	**h/aioSysDrv.h**

Table 4-5 **AIO Initialization Functions and Related Constants** *(Continued)*

Initialization Function	Configuration Parameter	Def. Value	Header File Constant used when arg = 0	Def. Value	Header File (all in *installDir*/target
	AIO_TASK_STACK_SIZE	0	**AIO_IO_STACK_DFLT**	0x7000	**h/aioSysDrv.h**

4.6.2 AIO Control Block

Each of the AIO calls takes an AIO control block (**aiocb**) as an argument to describe the AIO operation. The calling routine must allocate space for the control block, which is associated with a single AIO operation. No two concurrent AIO operations can use the same control block; an attempt to do so yields undefined results.

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

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_p
The address of VxWorks-specific data (non-POSIX).

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

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

4.6.3 Using AIO

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

After the I/O operations themselves execute, they also generate return values that reflect the success or failure of the I/O. There are two routines that you can use to get information about the success or failure of the I/O operation: **aio_error( )** and **aio_return( )**. You can use **aio_error( )** to get the status of an AIO operation (success, failure, or in progress), and **aio_return( )** to obtain the return values from the individual I/O operations. Until an AIO operation completes, its error status is **EINPROGRESS**. To cancel an AIO operation, call **aio_cancel( )**.

AIO with Periodic Checks for Completion

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

Example 4-2 **Asynchronous I/O**

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

/* includes */

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

/* defines */

#define BUFFER_SIZE 200

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

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

pipeDevCreate (exFile, 50, 100);

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

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

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

bzero ((char *) &aiocb_write, sizeof (struct aiocb));
    aiocb_write.aio_fildes = fd;
    aiocb_write.aio_buf = test_string;
    aiocb_write.aio_nbytes = strlen (test_string);
    aiocb_write.aio_reqprio = 0;
/* initiate the read */
if (aio_read (&aiocb_read) == -1)
    printf ("aioExample: aio_read failed\n");

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

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

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

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

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

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

Alternatives for Testing AIO Completion

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

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

The following example is similar to the preceding **aioExample( )**, except that it uses signals for notification that the write operation has finished. If you test this from the shell, spawn the routine to run at a lower priority than the AIO system tasks to assure that the test routine does not block completion of the AIO request.

Example 4-3 **Asynchronous I/O with Signals**

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

/* includes */

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

/* defines */

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

/* forward declarations */

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

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

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

pipeDevCreate (exFile, 50, 100);

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

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

/* set up signal handler for EXAMPLE_SIG_NO */
```

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

    /* initialize read and write aiocbs */

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

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

    /* set up signal info */

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

    /* initiate the read */

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

    /* verify that it is in progress */

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

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

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

    /* clean up */

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

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

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

    {
    /* print out what was read */
    printf ("mySigHandler: Got signal for aio write\n");

    /* write is complete so let's do cleanup for it here */
    if (aio_return (info->si_value.sival_ptr) == -1)
        {
        printf ("mySigHandler: aio_return for aiocb_write failed\n");
        printErrno (0);
        }
    }
```

4.7 Devices in VxWorks

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

Table 4-6 **Drivers Provided with VxWorks**

Module	Driver Description
ttyDrv	Terminal driver
ptyDrv	Pseudo-terminal driver
pipeDrv	Pipe driver
memDrv	Pseudo memory device driver
nfsDrv	NFS client driver
netDrv	Network driver for remote file access
ramDrv	RAM driver for creating a RAM disk
scsiLib	SCSI interface library
–	Other hardware-specific drivers

4.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)

VxWorks provides terminal and pseudo-terminal device drivers (*tty* and *pty* drivers). The *tty* driver is for actual terminals; the *pty* driver is for processes that simulate terminals. These pseudo terminals are useful in applications such as remote login facilities.

VxWorks serial I/O devices are buffered serial byte streams. Each device has a ring buffer (circular buffer) for both input and output. Reading from a *tty* device extracts bytes from the input ring. Writing to a *tty* device adds bytes to the output ring. The size of each ring buffer is specified when the device is created during system initialization.

NOTE: For the remainder of this section, the term *tty* is used to indicate both *tty* and *pty* devices

tty Options

The *tty* devices have a full range of options that affect the behavior of the device. These options are selected by setting bits in the device option word using the **ioctl( )** routine with the **FIOSETOPTIONS** function (see *I/O Control Functions*, p.135). For example, to set all the *tty* options except **OPT_MON_TRAP**:

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL & ~OPT_MON_TRAP);
```

Table 4-7 is a summary of the available options. The listed names are defined in the header file **ioLib.h**. For more detailed information, see the reference entry for **tyLib**.

Table 4-7 **Tty Options**

Library	Description
OPT_LINE	Selects *line mode*. (See *Raw Mode and Line Mode*, p.133.)
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 software flow control characters **CTRL+Q** and **CTRL+S** (**XON** and **XOFF**).
OPT_7_BIT	Strips the most significant bit from all input bytes.

Table 4-7 **Tty Options** *(Continued)*

Library	Description
OPT_MON_TRAP	Enables the special *ROM monitor trap* character, **CTRL+X** by default.
OPT_ABORT	Enables the special *target shell abort* character, **CTRL+C** by default. (Only useful if the target shell is configured into the system; see *6. Target Tools* in this manual for details.)
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. 132).

In *raw mode*, each input character is available to readers as soon as it is input from the device. Reading from a *tty* device in raw mode causes as many characters as possible to be extracted from the input ring, up to the limit of the user's read buffer. Input cannot be modified except as directed by other *tty* option bits.

In *line mode*, all input characters are saved until a **NEWLINE** character is input; then the entire line of characters, including the **NEWLINE**, is made available in the ring at one time. Reading from a *tty* device in line mode causes characters up to the end of the next line to be extracted from the input ring, up to the limit of the user's read buffer. Input can be modified by the special characters **CTRL+H** (backspace), **CTRL+U** (line-delete), and **CTRL+D** (end-of-file), which are discussed in *Tty Special Characters*, p. 133.

Tty Special Characters

The following special characters are enabled if the *tty* device operates in line mode, that is, with the **OPT_LINE** bit set:

- The backspace character, by default **CTRL+H**, causes successive previous characters to be deleted from the current line, up to the start of the line. It does this by echoing a backspace followed by a space, and then another backspace.
- The line-delete character, by default **CTRL+U**, deletes all the characters of the current line.

- The end-of-file (EOF) character, by default **CTRL+D**, causes the current line to become available in the input ring without a **NEWLINE** and without entering the EOF character itself. Thus if the EOF character is the first character typed on a line, reading that line returns a zero byte count, which is the usual indication of end-of-file.

The following characters have special effects if the *tty* device is operating with the corresponding option bit set:

- The software flow control characters **CTRL+Q** and **CTRL+S** (**XON** and **XOFF**). Receipt of a **CTRL+S** input character suspends output to that channel. Subsequent receipt of a **CTRL+Q** resumes the output. Conversely, when the VxWorks input buffer is almost full, a **CTRL+S** is output to signal the other side to suspend transmission. When the input buffer is empty enough, a **CTRL+Q** is output to signal the other side to resume transmission. The software flow control characters are enabled by **OPT_TANDEM**.
- The *ROM monitor trap* character, by default **CTRL+X**. This character traps to the ROM-resident monitor program. Note that this is drastic. All normal VxWorks functioning is suspended, and the computer system is controlled entirely by the monitor. Depending on the particular monitor, it may or may not be possible to restart VxWorks from the point of interruption.[1] The monitor trap character is enabled by **OPT_MON_TRAP**.
- The special *target shell abort* character, by default **CTRL+C**. This character restarts the target shell if it gets stuck in an unfriendly routine, such as one that has taken an unavailable semaphore or is caught in an infinite loop. The target shell abort character is enabled by **OPT_ABORT**.

The characters for most of these functions can be changed using the **tyLib** routines shown in Table 4-8.

Table 4-8 **Tty Special Characters**

Character	Description	Modifier
CTRL+H	backspace (character delete)	**tyBackspaceSet( )**
CTRL+U	line delete	**tyDeleteLineSet( )**
CTRL+D	EOF (end of file)	**tyEOFSet( )**
CTRL+C	target shell abort	**tyAbortSet( )**

[1] It will not be possible to restart VxWorks if un-handled external interrupts occur during the boot countdown.

Table 4-8 **Tty Special Characters** *(Continued)*

Character	Description	Modifier
CTRL+X	trap to boot ROMs	**tyMonitorTrapSet( )**
CTRL+S	output suspend	N/A
CTRL+Q	output resume	N/A

I/O Control Functions

The *tty* devices respond to the **ioctl( )** functions in Table 4-9, defined in **ioLib.h**. For more information, see the reference entries for **tyLib**, **ttyDrv**, and **ioctl( )**.

Table 4-9 **I/O Control Functions Supported by tyLib**

Function	Description
FIOBAUDRATE	Sets the baud rate to the specified argument.
FIOCANCEL	Cancels a read or write.
FIOFLUSH	Discards all bytes in the input and output buffers.
FIOGETNAME	Gets the filename of the *fd*.
FIOGETOPTIONS	Returns the current device option word.
FIONREAD	Gets the number of unread bytes in the input buffer.
FIONWRITE	Gets the number of bytes in the output buffer.
FIOSETOPTIONS	Sets the device option word.

⚠ **CAUTION:** To change the driver's hardware options (for example, the number of stop bits or parity bits), use the **ioctl( )** function **SIO_HW_OPTS_SET**. Because this command is not implemented in most drivers, you may need to add it to your BSP serial driver, which resides in *installDir***/target/src/drv/sio**. The details of how to implement this command depend on your board's serial chip. The constants defined in the header file *installDir***/target/h/sioLib.h** provide the POSIX definitions for setting the hardware options.

4.7.2 Pipe Devices

Pipes are virtual devices by which tasks communicate with each other through the I/O system. Tasks write messages to pipes; these messages can then be read by other tasks. Pipe devices are managed by **pipeDrv** and use the kernel message queue facility to bear the actual message traffic.

Creating Pipes

Pipes are created by calling the pipe create routine:

```
status = pipeDevCreate ("/pipe/name", maxMsgs, maxLength);
```

The new pipe can have at most *maxMsgs* messages queued at a time. Tasks that write to a pipe that already has the maximum number of messages queued are blocked until a message is dequeued. Each message in the pipe can be at most *maxLength* bytes long; attempts to write longer messages result in an error.

Writing to Pipes from ISRs

VxWorks pipes are designed to allow ISRs to write to pipes in the same way as task-level code. Many VxWorks facilities cannot be used from ISRs, including output to devices other than pipes. However, ISRs can use pipes to communicate with tasks, which can then invoke such facilities. ISRs write to a pipe using the **write( )** call. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message is discarded because the ISRs cannot pend. ISRs must not invoke any I/O function on pipes other than **write( )**. For more informationon ISRs, see *2.6 Interrupt Service Code: ISRs*, p.65.

I/O Control Functions

Pipe devices respond to the **ioctl( )** functions summarized in Table 4-10. The functions listed are defined in the header file **ioLib.h**. For more information, see the reference entries for **pipeDrv** and for **ioctl( )** in **ioLib**.

Table 4-10 **I/O Control Functions Supported by pipeDrv**

Function	Description
FIOFLUSH	Discards all messages in the pipe.

Table 4-10 **I/O Control Functions Supported by pipeDrv** *(Continued)*

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

4.7.3 Pseudo Memory Devices

The **memDrv** driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs. This driver does not implement a file system, unlike **ramDrv**. The **ramDrv** driver must be given memory over which it has absolute control; whereas **memDrv** provides a high-level method of reading and writing bytes in absolute memory locations through I/O calls.

Installing the Memory Driver

The driver is initialized automatically by the system with **memDrv( )** when the **INCLUDE_USR_MEMDRV** component is included in the VxWorks kernel domain. The call for device creation must be made from the kernel domain:

```
STATUS memDevCreate
    (char * name, char * base, int length)
```

Memory for the device is an absolute memory location beginning at *base*. The *length* parameter indicates the size of the memory. For additional information on the memory driver, see the entries for **memDrv( )**, **memDevCreate( )**, and **memDev CreateDir( )** in the *VxWorks API Reference*, as well as the entry for the host tool **memdrvbuild** in the *Tornado Tools* section of the online *Tornado Tools Reference*.

I/O Control Functions

The memory driver responds to the **ioctl( )** functions summarized in Table 4-11. The functions listed are defined in the header file **ioLib.h**.

Table 4-11 **I/O Control Functions Supported by memDrv**

Function	Description
FIOSEEK	Sets the current byte offset in the file.
FIOWHERE	Returns the current byte position in the file.

For more information, see the reference entries for **memDrv** and for **ioctl( )** in **ioLib**.

4.7.4 Network File System (NFS) Devices

Network File System (NFS) devices allow files on remote hosts to be accessed with the NFS protocol. The NFS protocol specifies both *client* software, to read files from remote machines, and *server* software, to export files to remote machines.

The driver **nfsDrv** acts as a VxWorks NFS client to access files on any NFS server on the network. VxWorks also allows you to run an NFS server to export files to other systems; see *VxWorks Network Programmer's Guide: File Access Applications*.

Using NFS devices, you can create, open, and access remote files exactly as though they were on a file system on a local disk. This is called *network transparency.*

Mounting a Remote NFS File System from VxWorks

Access to a remote NFS file system is established by mounting that file system locally and creating an I/O device for it using **nfsMount( )**. Its arguments are (1) the host name of the NFS server, (2) the name of the host file system, and (3) the local name for the file system.

For example, the following call mounts **/usr** of the host **mars** as **/vxusr** locally:

```
nfsMount ("mars", "/usr", "/vxusr");
```

This creates a VxWorks I/O device with the specified local name (**/vxusr**, in this example). If the local name is specified as NULL, the local name is the same as the remote name.

After a remote file system is mounted, the files are accessed as though the file system were local. Thus, after the previous example, opening the file **/vxusr/foo** opens the file **/usr/foo** on the host **mars**.

The remote file system must be *exported* by the system on which it actually resides. However, NFS servers can export only local file systems. Use the appropriate command on the server to see which file systems are local. NFS requires *authentication* parameters to identify the user making the remote access. To set these parameters, use the routines **nfsAuthUnixSet( )** and **nfsAuthUnixPrompt( )**.

To include NFS client support, use the **INCLUDE_NFS** component.

The subject of exporting and mounting NFS file systems and authenticating access permissions is discussed in more detail in *VxWorks Network Programmer's Guide: File Access Applications*. See also the reference entries **nfsLib** and **nfsDrv**, and the NFS documentation from Sun Microsystems.

I/O Control Functions for NFS Clients

NFS client devices respond to the **ioctl( )** functions summarized in Table 4-12. The functions listed are defined in **ioLib.h**. For more information, see the reference entries for **nfsDrv** and for **ioctl( )** in **ioLib**.

Table 4-12 **I/O Control Functions Supported by nfsDrv**

Function	Description
FIOFSTATGET	Gets file status information (directory entry data).
FIOGETNAME	Gets the filename of the *fd*.
FIONREAD	Gets the number of unread bytes in the file.
FIOREADDIR	Reads the next directory entry.
FIOSEEK	Sets the current byte offset in the file.
FIOSYNC	Flushes data to a remote NFS file.
FIOWHERE	Returns the current byte position in the file.

4.7.5 Non-NFS Network Devices

VxWorks also supports network access to files on a remote host through the Remote Shell protocol (RSH) or the File Transfer Protocol (FTP). These implementations of network devices use the driver **netDrv**. When a remote file is opened using RSH or FTP, the entire file is copied into local memory. As a result, the largest file that can be opened is restricted by the available memory. Read and write operations are performed on the memory-resident copy of the file. When closed, the file is copied back to the original remote file if it was modified.

In general, NFS devices are preferable to RSH and FTP devices for performance and flexibility, because NFS does not copy the entire file into local memory. However, NFS is not supported by all host systems.

Creating Network Devices

To access files on a remote host using either RSH or FTP, a network device must first be created by calling the routine **netDevCreate( )**. The arguments to **netDevCreate( )** are (1) the name of the device, (2) the name of the host the device accesses, and (3) which protocol to use: 0 (RSH) or 1 (FTP).

For example, the following call creates an RSH device called **mars:** that accesses the host **mars**. By convention, the name for a network device is the remote machine's name followed by a colon (**:**).

```
netDevCreate ("mars:", "mars", 0);
```

Files on a network device can be created, opened, and manipulated as if on a local disk. Thus, opening the file **mars:/usr/foo** actually opens **/usr/foo** on host **mars**.

Note that creating a network device allows access to any file or device on the remote system, while mounting an NFS file system allows access only to a specified file system.

For the files of a remote host to be accessible with RSH or FTP, permissions and user identification must be established on both the remote and local systems. Creating and configuring network devices is discussed in detail in *VxWorks Network Programmer's Guide: File Access Applications* and in the reference entry for **netDrv**.

I/O Control Functions

RSH and FTP devices respond to the same **ioctl( )** functions as NFS devices except for **FIOSYNC** and **FIOREADDIR**. The functions are defined in the header file **ioLib.h**. For more information, see the reference entries for **netDrv** and **ioctl( )**.

4.7.6 CBIO Interface

The core *cached block I/O* (CBIO) component, **INCLUDE_CBIO_MAIN**, provides an interface for file systems such as dosFs and rawFs. It is required for implementing these file systems.

The core CBIO component also provides generic services for other CBIO components that provide optional functionality. The optional CBIO components are:

INCLUDE_CBIO_DCACHE
Provides a disk cache.

INCLUDE_CBIO_DPART
Provides for partitioning volumes.

INCLUDE_CBIO_RAMDISK
Provides for RAM disks.

These components are discussed in the following sections as well as in the **cbioLib**, **dcacheCbio**, **dpartCbio**, and **ramDiskCbio** entries in the *VxWorks API Reference*.

CBIO Disk Cache

The CBIO disk cache is designed to make rotational media disk I/O more efficient, as well as to automatically detect changes in disks.

Disk I/O Efficiency

The disk cache reduces the number of disk accesses and accelerates disk read and write operations by means of the following techniques:

- Most Recently Used (MRU) disk blocks are stored in RAM, which results in the most frequently accessed data being retrieved from memory rather than from disk.
- Reading data from disk is performed in large units, relying on the read-ahead feature, one of the disk cache's tunable parameters.
- Write operations are optimized because they occur to memory first. Then, updating the disk happens in an orderly manner, by *delayed write*, another tunable parameter.

Overall, the main performance advantage arises from a dramatic reduction in the amount of time spent seeking by the disk drive, thus maximizing the time available for the disk to read and write actual data. In other words, you get efficient use of the disk drive's available throughput.

The disk cache offers a number of operational parameters that can be tuned by the user to suit a particular file system workload pattern, for example, delayed write, read ahead, and bypass threshold.

The technique of delaying writes to disk means that if the system is turned off unexpectedly, updates that have not yet been written to the disk are lost. To minimize the effect of a possible crash, the disk cache periodically updates the disk. Modified blocks of data are not kept in memory more than a specified period of time.

By specifying a small update period, the possible worst-case loss of data from a crash is the sum of changes possible during that specified period. For example, it is assumed that an update period of two seconds is sufficiently large to effectively optimize disk writes, yet small enough to make the potential loss of data a reasonably minor concern. It is possible to set the update period to zero, in which case, all updates are flushed to disk immediately.

The disk cache allows you to negotiate between disk performance and memory consumption: The more memory allocated to the disk cache, the higher the "hit ratio" observed, which means increasingly better performance of file system operations.

Bypass Threshold

Another tunable parameter is the *bypass threshold*, which defines how much data constitutes a request large enough to justify bypassing the disk cache.

When significantly large read or write requests are made by the application, the disk cache is circumvented and there is a direct transfer of data between the disk controller and the user data buffer. The use of bypassing, in conjunction with support for contiguous file allocation and access (the **FIOCONTIG ioctl( )** command and the **DOS_O_CONTIG open( )** flag), should provide performance equivalent to that offered by the raw file system — rawFs.

For details on all of the tunable parameters associated with the disk cache, see the entry for **dcacheDevTune( )** in the *VxWorks API Reference*. For complete details on the disk cache, see the entry for **dcacheCbio**.

Detection of Disk Changes

The disk cache component also provides for automatic detection of disk changes. The detection process is based on two assumptions: one about the uniqueness of a disk, and a second about the time required to change a disk.

The first assumption is that the first block—that is, the boot block of each cartridge or diskette—is unique. This is typically the case because most media is pre-formatted during manufacturing, and the boot block contains a 32-bit volume serial ID that is set by the manufacturer as unique. The formatting utility in dosFs preserves the volume ID when one exists on the volume being formatted; when no valid ID is found on the disk it creates a new one based on the time that the formatting takes place.

The second assumption is that it takes at least two seconds to physically replace a disk. If a disk has been inactive for two seconds or longer, its boot block is verified against a previously stored boot block signature. If they do not match, the file system module is notified of the change, and in turn un-mounts the volume, marking all previously open file descriptors as obsolete. A new volume is then mounted, if a disk is in the drive. This can trigger an optional automatic consistency check that detects any structural inconsistencies resulting from a previous disk removal in the midst of a disk update (see the entry for **dosFsDevCreate( )** in the *VxWorks API Reference*).

⚠ **CAUTION:** The disk cache is designed to benefit rotational media devices and should not be used with RAM disks and TrueFFS disks.

CBIO Disk Partition Handler

It has become commonplace to share fixed disks and removable cartridges between VxWorks target systems and PCs running Windows. Therefore, dosFs provides support for PC-style disk partitioning.

Two modules provide partition management mechanisms:

dpartCbioLib
This library implements a "generic" partition handling mechanism that is independent of any particular partition table format.

usrFdiskPartLib
This library decodes the partition tables found on most PCs (typically created with the **FDISK** utility). It is delivered in source form so as to enable customization of the partition table format. It also has a routine to create PC-style partitions (similar to **FDISK**). This release supports the creation of up to four partitions.

The **dpartCbioLib** module handles partitions on both fixed and removable drives, by calling the user-supplied partition table decode routine. The routine is called whenever removable media has been changed. When initializing the system, you must define the maximum number of partitions you expect to exist on a particular drive, and then associate a logical device name with each partition. If you insert removable media having fewer partitions than the maximum number defined, then only the number of partitions configured on the removable media are accessible. Additional partitions previously defined and named cannot be accessed.

CBIO RAM Disk

In some applications it is desirable to use a file system to organize and access data although no disk or other traditional media is present. The CBIO RAM disk facility allows the use of a file system to access data stored in RAM memory. RAM disks can be created using volatile as well a non-volatile RAM.

NOTE: The **ramDiskCbio** library implements a RAM disk using the CBIO interface; the **ramDrv** library implements a RAM disk using the **BLK_DEV** interface.

For more information, see the **ramDiskCbio** entries in the *VxWorks API Reference*.

I/O Control Functions for CBIO Devices

CBIO devices respond to the **ioctl( )** functions summarized in Table 4-13.

Table 4-13 **I/O Control Functions Supported by CBIO Devices**

Function	Description
CBIO_RESET	Resets the device.
CBIO_STATUS_CHK	Checks the device status.
CBIO_DEVICE_LOCK	Prevents disk removal.
CBIO_DEVICE_UNLOCK	Allows disk removal.
CBIO_DEVICE_EJECT	Dismounts the device.
CBIO_CACHE_FLUSH	Flushes dirty caches.
CBIO_CACHE_INVAL	Flushes and invalidates all.
CBIO_CACHE_NEWBLK	Allocates a scratch block.

4.7.7 Block Devices

A *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk. In VxWorks, the term *block* refers to the smallest addressable unit on the device. For most disk devices, a VxWorks block corresponds to a *sector*, although terminology varies.

Block devices in VxWorks have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, block device support consists of low-level drivers that interact with a file system. The file system, in turn, interacts with the I/O system. This arrangement allows a single low-level driver to be used with various different file systems and reduces the number of I/O functions that must be supported in the driver. The internal implementation of low-level drivers for block devices is discussed in *4.9.4 Block Devices*, p.176.

Block Device File Systems

For use with block devices, VxWorks is supplied with file system libraries compatible with the MS-DOS file systems. There are also libraries for a simple raw disk file system, for SCSI tape devices, for CD-ROM devices, and for flash memory devices. Use of these file systems is discussed in *5. Local File Systems* in this manual (which discusses the Target Server File System as well). Also see the entries for **dosFsLib**, **rawFsLib**, **tapeFsLib**, **cdromFsLib**, and **tffsDrv** in the *VxWorks API Reference*.

Block Device RAM Disk Drivers

RAM drivers, as implemented in **ramDrv**, emulate disk devices but actually keep all data in memory. Memory location and "disk" size are specified when a RAM device is created by calling **ramDevCreate( )**. This routine can be called repeatedly to create multiple RAM disks.

Memory for the RAM disk can be pre-allocated and the address passed to **ramDevCreate( )**, or memory can be automatically allocated from the system memory pool using **malloc( )**.

After the device is created, a name and file system (for example, dosFs or rawFs) must be associated with it using the file system's device creation routine and format routine; for example, **dosFsDevCreate( )** and **dosFsVolFormat( )**. Information describing the device is passed to the file system in a **BLK_DEV** structure. A pointer to this structure is returned by the RAM disk creation routine.

In the following example, a 200 KB RAM disk is created with automatically allocated memory, 512-byte sections, a single track, and no sector offset. The device is assigned the name **DEV1:** and initialized for use with dosFs.

```
BLK_DEV               *pBlkDev;
DOS_VOL_DESC          *pVolDesc;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
if (dosFsDevCreate ("DEV1:", pBlkDev, 20, NULL) == ERROR)
{
    printErrno( );
};

/* if you want to format then do the following */
if (dosFsVolFormat ("DEV1:", 2, NULL) == ERROR)
{
    printErrno( );
}
```

If the RAM disk memory already contains a disk image, the first argument to **ramDevCreate( )** is the address in memory, and the formatting arguments must be identical to those used when the image was created. For example:

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
if (dosFsDevCreate ("DEV1:", pBlkDev, 20, NULL) == ERROR)
{
    printErrno( );
}
```

In this case, only **dosFsDevCreate( )** must be used, because the file system already exists on the disk and does not require re-formatting. This procedure is useful if a RAM disk is to be created at the same address used in a previous boot of VxWorks. The contents of the RAM disk are then preserved.

NOTE: The **ramDiskCbio** library implements a RAM disk using the CBIO interface; the **ramDrv** library implements a RAM disk using the **BLK_DEV** interface.

For more information on RAM disk drivers, see the reference entry for **ramDrv**. For more information on file systems, see *5. Local File Systems*.

SCSI Drivers

SCSI is a standard peripheral interface that allows connection with a wide variety of hard disks, optical disks, floppy disks, tape drives, and CD-ROM devices. SCSI block drivers are compatible with the dosFs libraries, and offer several advantages for target configurations. They provide:

- local mass storage in non-networked environments
- faster I/O throughput than Ethernet networks

The SCSI-2 support in VxWorks supersedes previous SCSI support, although it offers the option of configuring the original SCSI functionality, now known as SCSI-1. With SCSI-2 enabled, the VxWorks environment can still handle SCSI-1 applications, such as file systems created under SCSI-1. However, applications that directly used SCSI-1 data structures defined in **scsiLib.h** may require modifications and recompilation for SCSI-2 compatibility.

The VxWorks SCSI implementation consists of two modules, one for the device-independent SCSI interface and one to support a specific SCSI controller. The **scsiLib** library provides routines that support the device-independent interface; device-specific libraries provide configuration routines that support specific controllers. There are also additional support routines for individual targets in **sysLib.c**.

Configuring SCSI Drivers

Components associated with SCSI drivers are listed in Table 4-14.

Table 4-14 **SCSI and Related Components**

Component	Description
INCLUDE_SCSI	Includes SCSI interface.
INCLUDE_SCSI2	Includes SCSI-2 extensions.
INCLUDE_SCSI_DMA	Enables DMA for SCSI.
INCLUDE_SCSI_BOOT	Allows booting from a SCSI device.
SCSI_AUTO_CONFIG	Auto-configures and locates all targets on a SCSI bus.
INCLUDE_DOSFS	Includes the DOS file system.
INCLUDE_TAPEFS	Includes the tape file system.
INCLUDE_CDROMFS	Includes CD-ROM file system support.

To include SCSI-1 functionality in VxWorks, use the **INCLUDE_SCSI** component. To include SCSI-2 functionality, you must use **INCLUDE_SCSI2** in addition to **INCLUDE_SCSI**.

Auto-configuration, DMA, and booting from a SCSI device are defined appropriately for each BSP. If you need to change these settings, see the reference for **sysScsiConfig( )** and the source file *installDir***/target/src/config/usrScsi.c**.

CAUTION: Including SCSI-2 in your VxWorks image can significantly increase the image size.

Configuring the SCSI Bus ID

Each board in a SCSI-2 environment must define a unique SCSI bus ID for the SCSI initiator. SCSI-1 drivers, which support only a single initiator at a time, assume an initiator SCSI bus ID of 7. However, SCSI-2 supports multiple initiators, up to eight initiators and targets at one time. Therefore, to ensure a unique ID, choose a value in the range 0-7 to be passed as a parameter to the driver's initialization routine (for example, **ncr710CtrlInitScsi2( )**) by the **sysScsiInit( )** routine in **sysScsi.c**. For more information, see the reference entry for the relevant driver initialization routine. If there are multiple boards on one SCSI bus, and all of these boards use the same BSP, then different versions of the BSP must be compiled for each board by assigning unique SCSI bus IDs.

ROM Size Adjustment for SCSI Boot

If the **INCLUDE_SCSI_BOOT** component is included, larger ROMs may be required for some boards.

Structure of the SCSI Subsystem

The SCSI subsystem supports libraries and drivers for both SCSI-1 and SCSI-2. It consists of the following six libraries which are independent of any SCSI controller:

scsiLib
: routines that provide the mechanism for switching SCSI requests to either the SCSI-1 library (**scsi1Lib**) or the SCSI-2 library (**scsi2Lib**), as configured by the board support package (BSP).

scsi1Lib
: SCSI-1 library routines and interface, used when only **INCLUDE_SCSI** is used (see *Configuring SCSI Drivers*, p.147).

scsi2Lib
: SCSI-2 library routines and all physical device creation and deletion routines.

scsiCommonLib
commands common to all types of SCSI devices.

scsiDirectLib
routines and commands for direct access devices (disks).

scsiSeqLib
routines and commands for sequential access block devices (tapes).

Controller-independent support for the SCSI-2 functionality is divided into **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, and **scsiSeqLib**. The interface to any of these SCSI-2 libraries can be accessed directly. However, **scsiSeqLib** is designed to be used in conjunction with tapeFs, while **scsiDirectLib** works with dosFs and rawFs. Applications written for SCSI-1 can be used with SCSI-2; however, SCSI-1 device drivers cannot.

VxWorks targets using SCSI interface controllers require a controller-specific device driver. These device drivers work in conjunction with the controller-independent SCSI libraries, and they provide controller configuration and initialization routines contained in controller-specific libraries. For example, the Western Digital WD33C93 SCSI controller is supported by the device driver libraries **wd33c93Lib**, **wd33c93Lib1**, and **wd33c93Lib2**. Routines tied to SCSI-1 (such as **wd33c93CtrlCreate( )**) and SCSI-2 (such as **wd33c93CtrlCreateScsi2( )**) are segregated into separate libraries to simplify configuration. There are also additional support routines for individual targets in **sysLib.c**.

Booting and Initialization

After VxWorks is built with SCSI support, the system startup code initializes the SCSI interface by executing **sysScsiInit( )** and **usrScsiConfig( )** when **INCLUDE_SCSI** is included in VxWorks. The call to **sysScsiInit( )** initializes the SCSI controller and sets up interrupt handling. The physical device configuration is specified in **usrScsiConfig( )**, which is in *installDir***/target/src/config/usrScsi.c**. The routine contains an example of the calling sequence to declare a hypothetical configuration, including:

- definition of physical devices with **scsiPhysDevCreate( )**
- creation of logical partitions with **scsiBlkDevCreate( )**
- specification of a file system with **dosFsDevCreate( )**.

If you are not using **SCSI_AUTO_CONFIG**, modify **usrScsiConfig( )** to reflect your actual configuration. For more information on the calls used in this routine, see the reference entries for **scsiPhysDevCreate( )**, **scsiBlkDevCreate( )**, **dosFsDevCreate( )**, and **dosFsVolFormat( )**.

Device-Specific Configuration Options

The SCSI libraries have the following default behaviors enabled:

- SCSI messages
- disconnects
- minimum period and maximum REQ/ACK offset
- tagged command queuing
- wide data transfer

Device-specific options do not need to be set if the device shares this default behavior. However, if you need to configure a device that diverges from these default characteristics, use **scsiTargetOptionsSet( )** to modify option values. These options are fields in the **SCSI_OPTIONS** structure, shown below. **SCSI_OPTIONS** is declared in **scsi2Lib.h**. You can choose to set some or all of these option values to suit your particular SCSI device and application.

```
typedef struct                    /* SCSI_OPTIONS - programmable options */
    {
    UINT      selTimeOut;         /* device selection time-out (us)       */
    BOOL      messages;           /* FALSE => do not use SCSI messages    */
    BOOL      disconnect;         /* FALSE => do not use disconnect       */
    UINT8      maxOffset;         /* max sync xfer offset (0 => async.)   */
    UINT8      minPeriod;         /* min sync xfer period (x 4 ns)        */
    SCSI_TAG_TYPE tagType;        /* default tag type                     */
    UINT      maxTags;            /* max cmd tags available (0 => untag   */
    UINT8      xferWidth;         /* wide data trnsfr width in SCSI units */
    } SCSI_OPTIONS;
```

There are numerous types of SCSI devices, each supporting its own mix of SCSI-2 features. To set device-specific options, define a **SCSI_OPTIONS** structure and assign the desired values to the structure's fields. After setting the appropriate fields, call **scsiTargetOptionsSet( )** to effect your selections. Example 4-5 illustrates one possible device configuration using **SCSI_OPTIONS**.

Call **scsiTargetOptionsSet( )** after initializing the SCSI subsystem, but before initializing the SCSI physical device. For more information about setting and implementing options, see the reference entry for **scsiTargetOptionsSet( )**.

⚠ **WARNING:** Calling **scsiTargetOptionsSet( )** after the physical device has been initialized may lead to undefined behavior.

The SCSI subsystem performs each SCSI command request as a SCSI transaction. This requires the SCSI subsystem to select a device. Different SCSI devices require different amounts of time to respond to a selection; in some cases, the **selTimeOut** field may need to be altered from the default.

If a device does not support SCSI messages, the boolean field **messages** can be set to FALSE. Similarly, if a device does not support disconnect/reconnect, the boolean field **disconnect** can be set to FALSE.

The SCSI subsystem automatically tries to negotiate synchronous data transfer parameters. However, if a SCSI device does not support synchronous data transfer, set the **maxOffset** field to 0. By default, the SCSI subsystem tries to negotiate the maximum possible REQ/ACK offset and the minimum possible data transfer period supported by the SCSI controller on the VxWorks target. This is done to maximize the speed of transfers between two devices. However, speed depends upon electrical characteristics, like cable length, cable quality, and device termination; therefore, it may be necessary to reduce the values of **maxOffset** or **minPeriod** for fast transfers.

The **tagType** field defines the type of tagged command queuing desired, using one of the following macros:

– **SCSI_TAG_UNTAGGED**
– **SCSI_TAG_SIMPLE**
– **SCSI_TAG_ORDERED**
– **SCSI_TAG_HEAD_OF_QUEUE**

For more information about the types of tagged command queuing available, see the ANSI X3T9-I/O Interface Specification *Small Computer System Interface (SCSI-2)*.

The **maxTags** field sets the maximum number of command tags available for a particular SCSI device.

Wide data transfers with a SCSI target device are automatically negotiated upon initialization by the SCSI subsystem. Wide data transfer parameters are always negotiated before synchronous data transfer parameters, as specified by the SCSI ANSI specification, because a wide negotiation resets any prior negotiation of synchronous parameters. However, if a SCSI device does not support wide parameters and there are problems initializing that device, you must set the **xferWidth** field to 0. By default, the SCSI subsystem tries to negotiate the maximum possible transfer width supported by the SCSI controller on the VxWorks target in order to maximize the default transfer speed between the two devices. For more information on the actual routine call, see the reference entry for **scsiTargetOptionsSet( )**.

SCSI Configuration Examples

The following examples show some possible configurations for different SCSI devices. Example 4-4 is a simple block device configuration setup. Example 4-5 involves selecting special options and demonstrates the use of **scsiTargetOptionsSet( )**. Example 4-6 configures a tape device and a tape file system. Example 4-7 configures a SCSI device for synchronous data transfer. Example 4-8 shows how to configure the SCSI bus ID. These examples can be embedded either in the **usrScsiConfig( )** routine or in a user-defined SCSI configuration function.

Example 4-4 **Configuring SCSI Drivers**

In the following example, **usrScsiConfig( )** was modified to reflect a new system configuration. The new configuration has a SCSI disk with a bus ID of 4 and a Logical Unit Number (LUN) of 0 (zero). The disk is configured with a dosFs file system (with a total size of 0x20000 blocks) and a rawFs file system (spanning the remainder of the disk). The following **usrScsiConfig( )** code reflects this modification.

```
/* configure Winchester at busId = 4, LUN = 0 */

if ((pSpd40 = scsiPhysDevCreate (pSysScsiCtrl, 4, 0, 0, NONE, 0, 0, 0))
        == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    }
else
    {
    /* create block devices - one for dosFs and one for rawFs */

    if (((pSbd0 = scsiBlkDevCreate (pSpd40, 0x20000, 0)) == NULL) ||
        ((pSbd1 = scsiBlkDevCreate (pSpd40, 0, 0x20000)) == NULL))
        {
        return (ERROR);
        }

    /* initialize both dosFs and rawFs file systems */

    if ((dosFsDevInit ("/sd0/", pSbd0, NULL) == NULL) ||
        (rawFsDevInit ("/sd1/", pSbd1) == NULL))
        {
        return (ERROR);
        }
    }
```

If problems with your configuration occur, insert the following lines at the beginning of **usrScsiConfig( )** to obtain further information on SCSI bus activity.

```
    #if FALSE
    scsiDebug = TRUE;
    scsiIntsDebug = TRUE;
    #endif
```

Do not declare the global variables **scsiDebug** and **scsiIntsDebug** locally. They can be set or reset from the shell; see the *Tornado User's Reference: Shell* for details.

Example 4-5 **Configuring a SCSI Disk Drive with Asynchronous Data Transfer and No Tagged Command Queuing**

In this example, a SCSI disk device is configured without support for synchronous data transfer and tagged command queuing. The **scsiTargetOptionsSet( )** routine is used to turn off these features. The SCSI ID of this disk device is 2, and the LUN is 0:

```
int                 which;
SCSI_OPTIONS        option;
int                 devBusId;

devBusId = 2;
which = SCSI_SET_OPT_XFER_PARAMS | SCSI_SET_OPT_TAG_PARAMS;
option.maxOffset = SCSI_SYNC_XFER_ASYNC_OFFSET;
                                            /* => 0 defined in scsi2Lib.h */
option.minPeriod = SCSI_SYNC_XFER_MIN_PERIOD;  /* defined in scsi2Lib.h */
option.tagType = SCSI_TAG_UNTAGGED;      /* defined in scsi2Lib.h */
option.maxTag = SCSI_MAX_TAGS;

if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId, &option, which) == ERROR)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n", 0, 0, 0, 0,
        0, 0);
    return (ERROR);
    }

/* configure SCSI disk drive at busId = devBusId, LUN = 0 */

if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE, 0, 0,
        0)) == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }
```

Example 4-6 **Working with Tape Devices**

SCSI tape devices can be controlled using common commands from **scsiCommonLib** and sequential commands from **scsiSeqLib**. These commands use a pointer to a SCSI sequential device structure, **SEQ_DEV**, defined in **seqIo.h**. For more information on controlling SCSI tape devices, see the reference entries for these libraries.

This example configures a SCSI tape device whose bus ID is 5 and whose LUN is 0. It includes commands to create a physical device pointer, set up a sequential device, and initialize a tapeFs device.

```
/* configure Exabyte 8mm tape drive at busId = 5, LUN = 0 */
if ((pSpd50 = scsiPhysDevCreate (pSysScsiCtrl, 5, 0, 0, NONE, 0, 0, 0))
        == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }

/* configure the sequential device for this physical device */
if ((pSd0 = scsiSeqDevCreate (pSpd50)) == (SEQ_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiSeqDevCreate failed.\n");
        return (ERROR);
    }

/* setup the tape device configuration */
pTapeConfig = (TAPE_CONFIG *) calloc (sizeof (TAPE_CONFIG), 1);
pTapeConfig->rewind = TRUE;   /* this is a rewind device */
pTapeConfig->blkSize = 512;   /* uses 512 byte fixed blocks */

/* initialize a tapeFs device */
if (tapeFsDevInit ("/tape1", pSd0, pTapeConfig) == NULL)
    {
    return (ERROR);
    }

/* rewind the physical device using scsiSeqLib interface directly*/
if (scsiRewind (pSd0) == ERROR)
    {
    return (ERROR);
    }
```

Example 4-7 **Configuring a SCSI Disk for Synchronous Data Transfer with Non-Default Offset and Period Values**

In this example, a SCSI disk drive is configured with support for synchronous data transfer. The offset and period values are user-defined and differ from the driver default values. The chosen period is 25, defined in SCSI units of 4 ns. Thus, the period is actually 4 * 25 = 100 ns. The synchronous offset is chosen to be 2. Note that you may need to adjust the values depending on your hardware environment.

```
int                     which;
SCSI_OPTIONS            option;
int                     devBusId;

devBusId = 2;

    which = SCSI_SET_IPT_XFER_PARAMS;
    option.maxOffset = 2;
    option.minPeriod = 25;

    if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId &option, which) ==
        ERROR)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n",
                         0, 0, 0, 0, 0, 0)
        return (ERROR);
        }

    /* configure SCSI disk drive at busId = devBusId, LUN = 0 */

    if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE,
                                     0, 0, 0)) == (SCSI_PHYS_DEV *) NULL)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n")
        return (ERROR);
        }
```

Example 4-8 **Changing the Bus ID of the SCSI Controller**

To change the bus ID of the SCSI controller, modify **sysScsiInit( )** in **sysScsi.c**. Set the SCSI bus ID to a value between 0 and 7 in the call to *xxx***CtrlInitScsi2( )**, where *xxx* is the controller name. The default bus ID for the SCSI controller is 7.

Troubleshooting

- **Incompatibilities Between SCSI-1 and SCSI-2**

  Applications written for SCSI-1 may not execute for SCSI-2 because data structures in **scsi2Lib.h**, such as **SCSI_TRANSACTION** and **SCSI_PHYS_DEV**, have changed. This applies only if the application used these structures directly.

  If this is the case, you can choose to configure only the SCSI-1 level of support, or you can modify your application according to the data structures in **scsi2Lib.h**. In order to set new fields in the modified structure, some applications may simply need to be recompiled, and some applications will have to be modified and then recompiled.

- **SCSI Bus Failure**

  If your SCSI bus hangs, it could be for a variety of reasons. Some of the more common are:

  – Your cable has a defect. This is the most common cause of failure.

  – The cable exceeds the cumulative maximum length of 6 meters specified in the SCSI-2 standard, thus changing the electrical characteristics of the SCSI signals.

  – The bus is not terminated correctly. Consider providing termination power at both ends of the cable, as defined in the SCSI-2 ANSI specification.

- The minimum transfer period is insufficient or the REQ/ACK offset is too great. Use **scsiTargetOptionsSet( )** to set appropriate values for these options.
- The driver is trying to negotiate wide data transfers on a device that does not support them. In rejecting wide transfers, the device-specific driver cannot handle this phase mismatch. Use **scsiTargetOptionsSet( )** to set the appropriate value for the **xferWidth** field for that particular SCSI device.

4.7.8 Sockets

In VxWorks, the underlying basis of network communications is *sockets*. A socket is an endpoint for communication between tasks; data is sent from one socket to another. Sockets are not created or opened using the standard I/O functions. Instead, they are created by calling **socket( )**, and connected and accessed using other routines in **sockLib**. However, after a *stream* socket (using TCP) is created and connected, it can be accessed as a standard I/O device, using **read( )**, **write( )**, **ioctl( )**, and **close( )**. The value returned by **socket( )** as the socket handle is in fact an I/O system file descriptor (*fd*).

VxWorks socket routines are source-compatible with the BSD 4.4 UNIX socket functions and the Windows Sockets (Winsock 1.1) networking standard. Use of these routines is discussed in *VxWorks Network Programmer's Guide: Networking APIs*.

4.8 Differences Between VxWorks and Host System I/O

Most commonplace uses of I/O in VxWorks are completely source-compatible with I/O in UNIX and Windows. However, note the following differences:

- **Device Configuration.** In VxWorks, device drivers can be installed and removed dynamically.
- **File Descriptors.** In UNIX and Windows, *fd*s are unique to each process. In VxWorks, *fd*s are global entities, accessible by any task, except for standard input, standard output, and standard error (0, 1, and 2), which can be task specific.

- **I/O Control.** The specific parameters passed to **ioctl( )** functions can differ between UNIX and VxWorks.
- **Driver Routines.** In UNIX, device drivers execute in system mode and cannot be preempted. In VxWorks, driver routines can be preempted because they execute within the context of the task that invoked them.

4.9 Internal Structure

The VxWorks I/O system differs from most I/O systems in the way that the work of performing user I/O requests is distributed between the device-independent I/O system and the device drivers themselves.

In many systems, the device driver supplies a few routines to perform low-level I/O functions such as reading a sequence of bytes from, or writing them to, character-oriented devices. The higher-level protocols, such as communications protocols on character-oriented devices, are implemented in the device-independent part of the I/O system. The user requests are heavily processed by the I/O system before the driver routines get control.

While this approach is designed to make it easy to implement drivers and to ensure that devices behave as much alike as possible, it has several drawbacks. The driver writer is often seriously hampered in implementing alternative protocols that are not provided by the existing I/O system. In a real-time system, it is sometimes desirable to bypass the standard protocols altogether for certain devices where throughput is critical, or where the device does not fit the standard model.

In the VxWorks I/O system, minimal processing is done on user I/O requests before control is given to the device driver. The VxWorks I/O system acts as a switch to route user requests to appropriate driver-supplied routines. Each driver can then process the raw user requests as appropriate to its devices. In addition, however, several high-level subroutine libraries are available to driver writers that implement standard protocols for both character- and block-oriented devices. Thu,s 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 4-8). Block devices are used for storing file systems. They are random access devices where data is transferred in blocks. Examples of block devices include hard and floppy disks. Character devices are any device that does not fall in the block category. Examples of character devices include serial and graphical input devices, for example, terminals and graphics tablets.

As discussed in earlier sections, the three main elements of the VxWorks I/O system are drivers, devices, and files. The following sections describe these elements in detail. The discussion focuses on character drivers; however, much of it is applicable for block devices. Because block drivers must interact with VxWorks file systems, they use a slightly different organization; see *4.9.4 Block Devices*, p. 176.

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. It is not a complete text on writing a device driver. For detailed information on this subject, see the *VxWorks BSP Developer's Guide*.

Example 4-9 shows the abbreviated code for a hypothetical driver that is used as an example throughout the following discussions. This example driver is typical of drivers for character-oriented devices.

In VxWorks, each driver has a short, unique abbreviation, such as **net** or **tty**, which is used as a prefix for each of its routines. The abbreviation for the example driver is *xx*.

Example 4-9 **Hypothetical Driver**

```
/*
 * xxDrv - driver initialization routine
 * xxDrv() init's the driver. It installs the driver via iosDrvInstall.
 * It may allocate data structures, connect ISRs, and initialize hardware
*/

STATUS xxDrv ()
  {
  xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl)
;
  (void) intConnect (intvec, xxInterrupt, ...);
  ...
  }

/*************************************************************************
 * xxDevCreate - device creation routine
 *
 * Called to add a device called <name> to be svced by this driver. Other
 * driver-dependent arguments may include buffer sizes, device addresses.
 * The routine adds the device to the I/O system by calling iosDevAdd.
 * It may also allocate and initialize data structures for the device,
 * initialize semaphores, initialize device hardware, and so on.
 */

STATUS xxDevCreate (name, ...)
  char * name;
  ...
  {
  status = iosDevAdd (xxDev, name, xxDrvNum);
  ...
  }

/*
 *
 * The following routines implement the basic I/O functions.
 * The xxOpen() return value is meaningful only to this driver,
 * and is passed back as an argument to the other I/O routines.
 */

int xxOpen (xxDev, remainder, mode)
  XXDEV * xxDev;
  char * remainder;
  int mode;
  {
  /* serial devices should have no file name part */

  if (remainder[0] != 0)
    return (ERROR);
  else
    return ((int) xxDev);
  }

int xxRead (xxDev, buffer, nBytes)
  XXDEV * xxDev;
  char * buffer;
  int nBytes;
  ...
int xxWrite (xxDev, buffer, nBytes)
  ...
int xxIoctl (xxDev, requestCode, arg)
  ...

/*
 * xxInterrupt - interrupt service routine
 *
 * Most drivers have routines that handle interrupts from the devices
 * serviced by the driver. These routines are connected to the interrupts
 * by calling intConnect (usually in xxDrv above). They can receive a
 * single argument, specified in the call to intConnect (see intLib).
*/

VOID xxInterrupt (arg)
  ...
```

4.9.1 Drivers

A driver for a non-block device implements the seven basic I/O functions—**creat( )**, **delete( )**, **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**—for a particular kind of device. In general, this type of driver has routines that implement each of these functions, although some of the routines can be omitted if the functions are not operative with that device.

Drivers can optionally allow tasks to wait for activity on multiple file descriptors. This is implemented using the driver's **ioctl( )** routine; see *Implementing select( )*, p. 168.

A driver for a block device interfaces with a file system, rather than directly with the I/O system. The file system in turn implements most I/O functions. The driver need only supply routines to read and write blocks, reset the device, perform I/O control, and check device status. Drivers for block devices have a number of special requirements that are discussed in *4.9.4 Block Devices*, p. 176.

When the user invokes one of the basic I/O functions, the I/O system routes the request to the appropriate routine of a specific driver, as detailed in the following sections. The driver's routine runs in the calling task's context, as though it were called directly from the application. Thus, the driver is free to use any facilities normally available to tasks, including I/O to other devices. This means that most drivers have to use some mechanism to provide mutual exclusion to critical regions of code. The usual mechanism is the semaphore facility provided in **semLib**.

In addition to the routines that implement the seven basic I/O functions, drivers also have three other routines:

- An initialization routine that installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization. This routine is typically named *xx***Drv( )**.
- A routine to add devices that are to be serviced by the driver to the I/O system. This routine is typically named *xx***DevCreate( )**.
- Interrupt-level routines that are connected to the interrupts of the devices serviced by the driver.

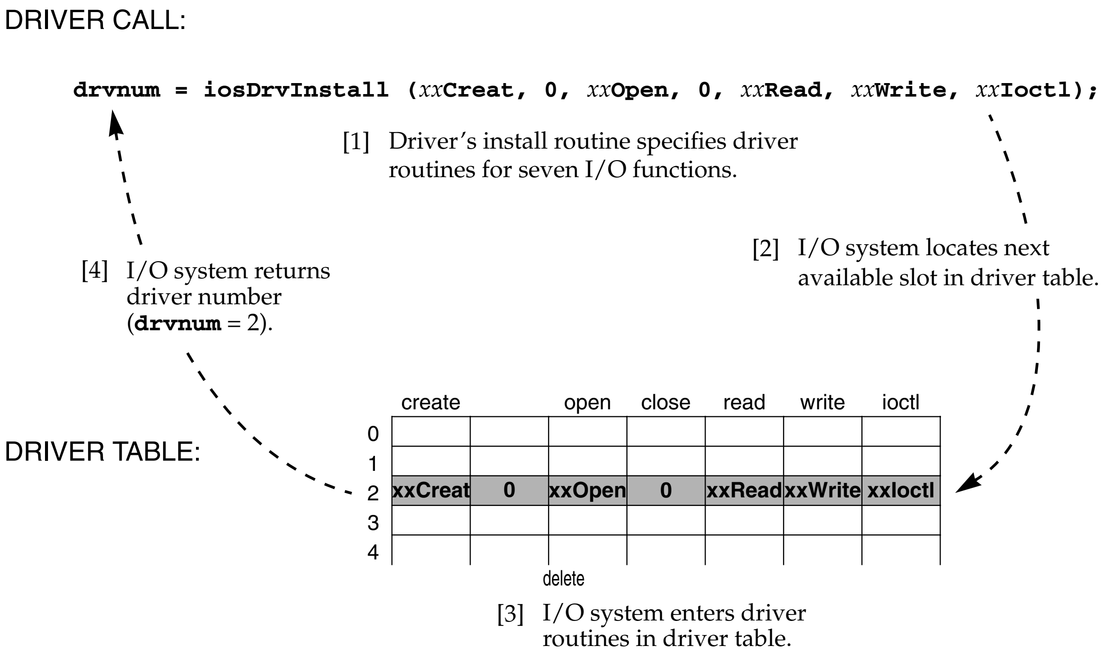
The Driver Table and Installing Drivers

The function of the I/O system is to route user I/O requests to the appropriate routine of the appropriate driver. The I/O system does this by maintaining a table that contains the address of each routine for each driver. Drivers are installed dynamically by calling the I/O system internal routine **iosDrvInstall( )**. The arguments to this routine are the addresses of the seven I/O routines for the new driver. The **iosDrvInstall( )** routine enters these addresses in a free slot in the driver table and returns the index of this slot. This index is known as the *driver number* and is used subsequently to associate particular devices with the driver.

Null (0) addresses can be specified for some of the seven routines. This indicates that the driver does not process those functions. For non-file-system drivers, **close( )** and **delete( )** often do nothing as far as the driver is concerned.

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 4-2 **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.

[3] I/O system enters driver routines 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							

Example of Installing a Driver

Figure 4-2 shows the actions taken by the example driver and by the I/O system when the initialization routine *xx***Drv( )** runs.

The driver calls **iosDrvInstall( )**, specifying the addresses of the driver's routines for the seven basic I/O functions. Then, the I/O system:

1. Locates the next available slot in the driver table, in this case slot 2.
2. Enters the addresses of the driver routines in the driver table.
3. Returns the slot number as the driver number of the newly installed driver.

4.9.2 Devices

Some drivers are capable of servicing many instances of a particular kind of device. For example, a single driver for a serial communications device can often handle many separate channels that differ only in a few parameters, such as device address.

In the VxWorks I/O system, devices are defined by a data structure called a *device header* (**DEV_HDR**). This data structure contains the device name string and the driver number for the driver that services this device. The device headers for all the devices in the system are kept in a memory-resident linked list called the *device list*. The device header is the initial part of a larger structure determined by the individual drivers. This larger structure, called a *device descriptor*, contains additional device-specific data such as device addresses, buffers, and semaphores.

The Device List and Adding Devices

Non-block devices are added to the I/O system dynamically by calling the internal I/O routine **iosDevAdd( )**. The arguments to **iosDevAdd( )** are the address of the device descriptor for the new device, the device's name, and the driver number of the driver that services the device. The device descriptor specified by the driver can contain any necessary device-dependent information, as long as it begins with a device header. The driver does not need to fill in the device header, only the device-dependent information. The **iosDevAdd( )** routine enters the specified device name and the driver number in the device header and adds it to the system device list.

To add a block device to the I/O system, call the device initialization routine for the file system required on that device (**dosFsDevCreate( )** or **rawFsDevInit( )**). The device initialization routine then calls **iosDevAdd( )** automatically.

The routine **iosDevFind( )** can be used to locate the device structure (by obtaining a pointer to the **DEV_HDR**, which is the first member of that structure) and to verify that a device name exists in the table. 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.
         */
        }
```

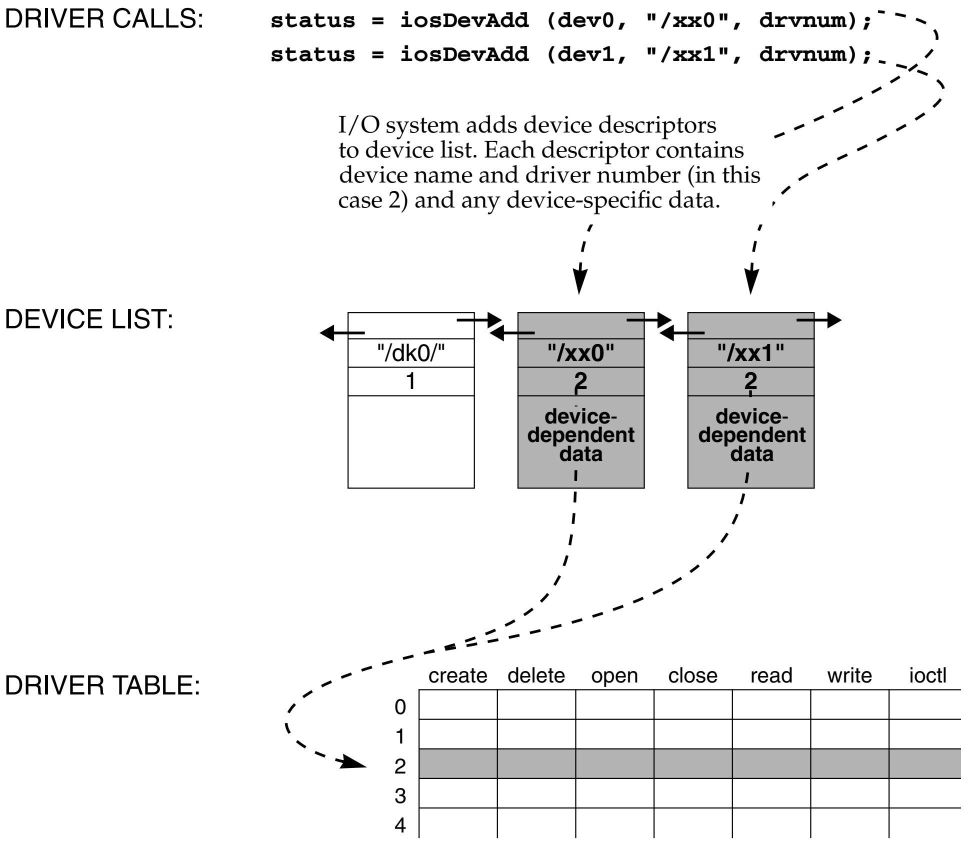
Example of Adding Devices

In Figure 4-3, the example driver's device creation routine *xx***DevCreate( )** adds devices to the I/O system by calling **iosDevAdd( )**.

4.9.3 File Descriptors

Several *fd*s can be open to a single device at one time. A device driver can maintain additional information associated with an *fd* beyond the I/O system's device information. In particular, devices on which multiple files can be open at one time have file-specific information (for example, file offset) associated with each *fd*. You can also have several *fd*s open to a non-block device, such as a *tty*; typically there is no additional information, and thus writing on any of the *fd*s produces identical results.

Figure 4-3 **Example – Addition of Devices to I/O System**



```
status = iosDevAdd (dev0, "/xx0", drvnum);
status = iosDevAdd (dev1, "/xx1", drvnum);
```

DRIVER CALLS:
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

The Fd Table

Files are opened with **open( )** (or **creat( )**). The I/O system searches the device list for a device name that matches the filename (or an initial substring) specified by the caller. If a match is found, the I/O system uses the driver number contained in the corresponding device header to locate and call the driver's open routine in the driver table.

The I/O system must establish an association between the file descriptor used by the caller in subsequent I/O calls, and the driver that services it. Additionally, the driver must associate some data structure per descriptor. In the case of non-block devices, this is usually the device descriptor that was located by the I/O system.

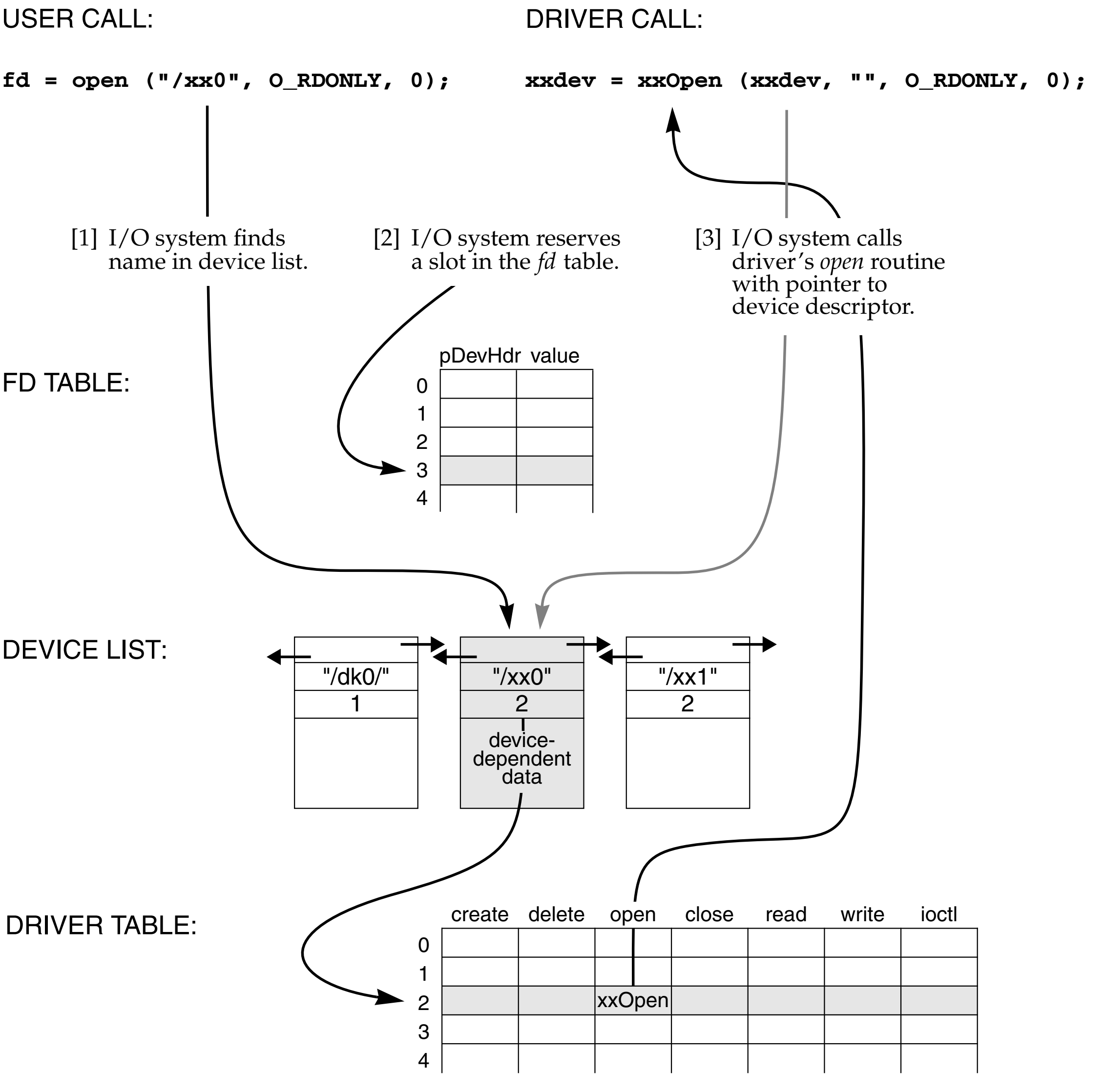
The I/O system maintains these associations in a table called the *fd table*. This table contains the driver number and an additional driver-determined 4-byte value. The driver value is the internal descriptor returned by the driver's open routine, and can be any nonnegative value the driver requires to identify the file. In subsequent calls to the driver's other I/O functions (**read( )**, **write( )**, **ioctl( )**, and **close( )**), this value is supplied to the driver in place of the *fd* in the application-level I/O call.

Example of Opening a File

In Figure 4-4 and Figure 4-5, 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 *fd* table and creates a new file descriptor object, which is used if the open is successful.
3. It then looks up the address of the driver's open routine, *xx***Open( )**, and calls that routine. Note that the arguments to *xx***Open( )** are transformed by the I/O system from the user's original arguments to **open( )**. The first argument to *xx***Open( )** is a pointer to the device descriptor the I/O system located in the full filename search. The next parameter is the *remainder* of the filename specified by the user, after removing the initial substring that matched the device name. In this case, because the device name matched the entire filename, the remainder passed to the driver is a null string. The driver is free to interpret this remainder in any way it wants. In the case of block devices, this remainder is the name of a file on the device. In the case of non-block devices like this one, it is usually an error for the remainder to be anything *but* the null string. The third parameter is the file access flag, in this case **O_RDONLY**; that is, the file is opened for reading only. The last parameter is the mode, as passed to the original **open( )** routine.
4. It executes *xx***Open( )**, which returns a value that subsequently identifies the newly opened file. In this case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the file being opened. Note that if the driver returns only the device descriptor, the driver cannot distinguish multiple files opened to the same device. In the case of non-block device drivers, this is usually appropriate.
5. The I/O system then enters the driver number and the value returned by *xx***Open( )** in the new file descriptor object. .

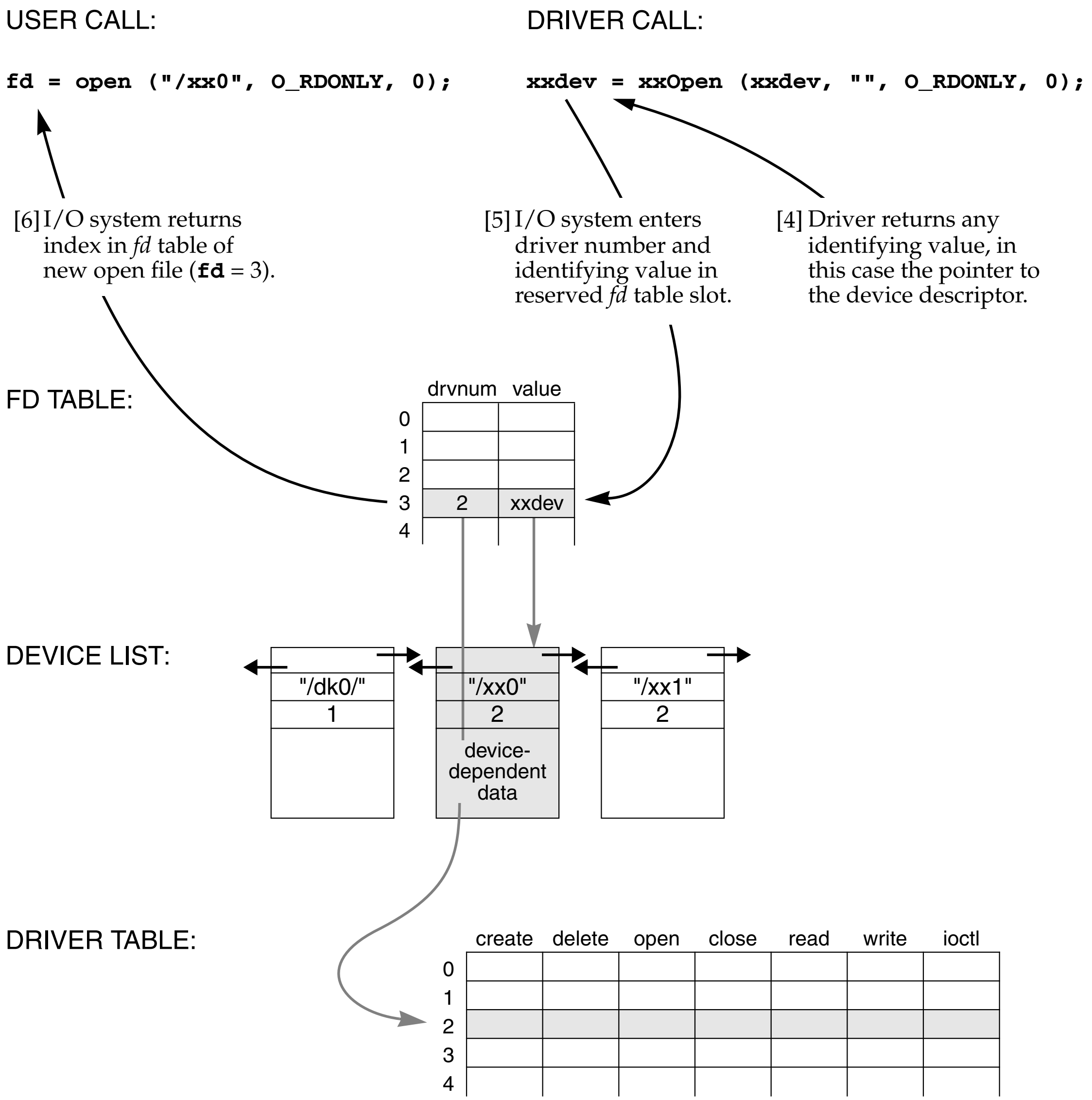
Figure 4-4 **Example: Call to I/O Routine open( ) [Part 1]**


USER CALL:
DRIVER CALL:
fd = open ("/xx0", O_RDONLY, 0);
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
[1] I/O system finds name in device list.
[2] I/O system reserves a slot in the fd table.
[3] I/O system calls driver's open routine with pointer to device descriptor.
FD 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

Again, the value entered in the *fd* 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 *fd* table, in this case 3.

Figure 4-5 **Example: Call to I/O Routine open( ) [Part 2]**


USER CALL:
fd = open ("/xx0", O_RDONLY, 0);
DRIVER CALL:
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
[6] I/O system returns index in fd table of new open file (fd = 3).
[5] I/O system enters driver number and identifying value in reserved fd table slot.
[4] Driver returns any identifying value, in this case the pointer to the device descriptor.
FD 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 4-6, the user calls **read( )** to obtain input data from the file. The specified *fd* is the index into the *fd* table for this file. The I/O system uses the driver number contained in the table to locate the driver's read routine, *xx***Read( )**. The I/O system calls *xx***Read( )**, passing it the identifying value in the *fd* table that was returned by the driver's open routine, *xx***Open( )**. Again, in this case the value is the pointer to the device descriptor. The driver's read routine then does whatever is necessary to read data from the device. The process for user calls to **write( )** and **ioctl( )** follow the same procedure.

Example of Closing a File

The user terminates the use of a file by calling **close( )**. As in the case of **read( )**, the I/O system uses the driver number contained in the *fd* table to locate the driver's close routine. In the example driver, no close routine is specified; thus no driver routines are called. Instead, the I/O system marks the slot in the *fd* table as being available. Any subsequent references to that *fd* cause an error. However, subsequent calls to **open( )** can reuse that slot.

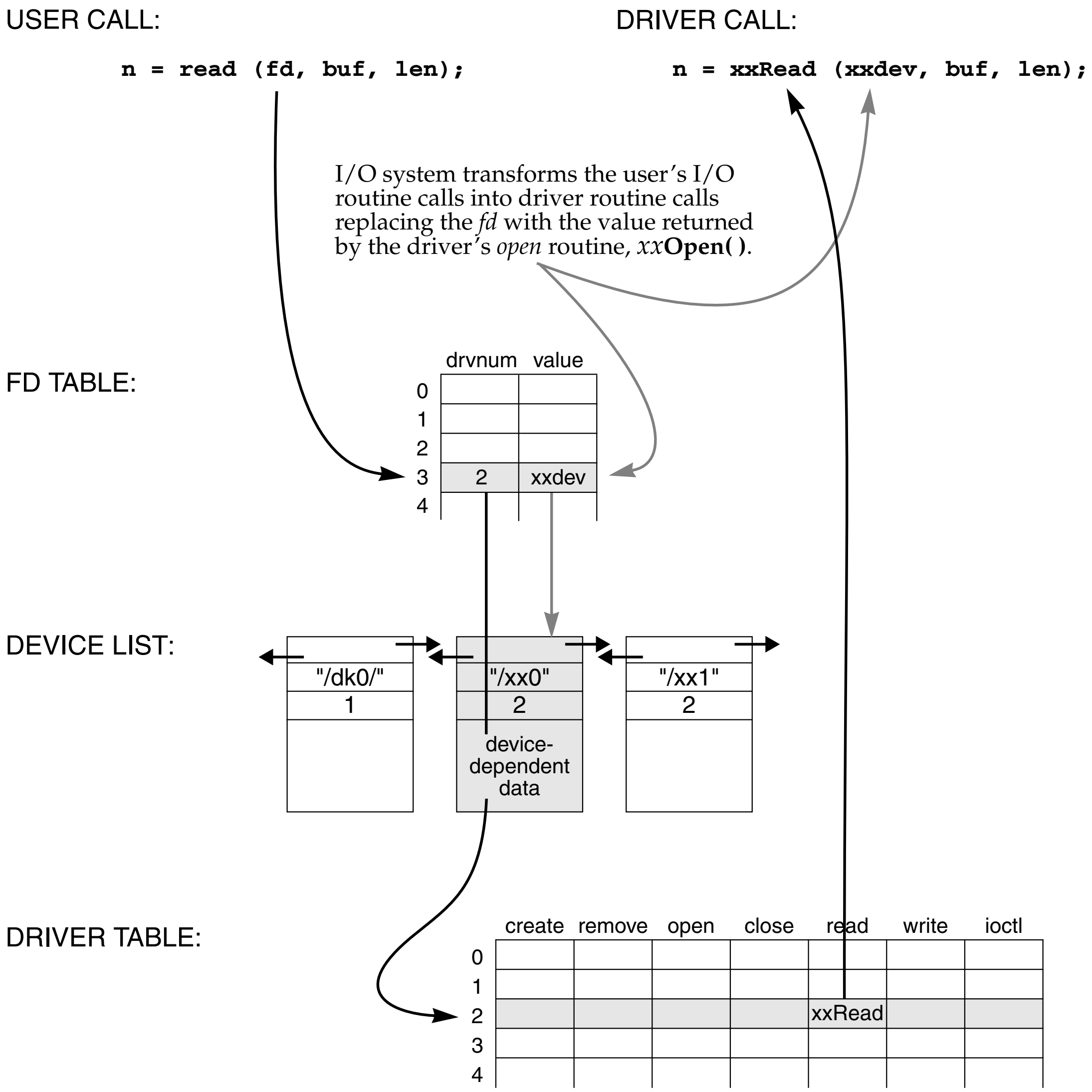
Implementing select( )

Supporting **select( )** in your driver allows tasks to wait for input from multiple devices or to specify a maximum time to wait for the device to become ready for I/O. Writing a driver that supports **select( )** is simple, because most of the 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.

Figure 4-6 **Example: Call to I/O Routine read( )**



I/O system transforms the user's I/O routine calls into driver routine calls replacing the *fd* with the value returned by the driver's *open* routine, *xx***Open( )**.

For a device driver to support **select( )**, it must declare a **SEL_WAKEUP_LIST** structure (typically declared as part of the device descriptor structure) and initialize it by calling **selWakeupListInit( )**. This is done in the driver's *xx***DevCreate( )** routine. When a task calls **select( )**, **selectLib** calls the driver's **ioctl( )** routine with the function **FIOSELECT** or **FIOUNSELECT**. If **ioctl( )** is called with **FIOSELECT,** the driver must do the following:

1. Add the **SEL_WAKEUP_NODE** (provided as the third argument of **ioctl( )**) to the **SEL_WAKEUP_LIST** by calling **selNodeAdd( )**.
2. Use the routine **selWakeupType( )** to check whether the task is waiting for data to read from the device (**SELREAD**) or if the device is ready to be written (**SELWRITE**).
3. If the device is ready (for reading or writing as determined by **selWakeupType( )**), the driver calls the routine **selWakeup( )** to make sure that the **select( )** call in the task does not pend. This avoids the situation where the task is blocked but the device is ready.

If **ioctl( )** is called with **FIOUNSELECT**, the driver calls **selNodeDelete( )** to remove the provided **SEL_WAKEUP_NODE** from the wakeup list.

When the device becomes available, **selWakeupAll( )** is used to unblock all the tasks waiting on this device. Although this typically occurs in the driver's ISR, it can also occur elsewhere. For example, a pipe driver might call **selWakeupAll( )** from its *xx***Read( )** routine to unblock all the tasks waiting to write, now that there is room in the pipe to store the data. Similarly the pipe's *xx***Write( )** routine might call **selWakeupAll( )** to unblock all the tasks waiting to read, now that there is data in the pipe.

Example 4-10 **Driver Code Using the Select Facility**

```
/* This code fragment shows how a driver might support select(). In this
 * example, the driver unblocks tasks waiting for the device to become ready
 * in its interrupt service routine.
 */

/* myDrvLib.h - header file for driver */

typedef struct     /* MY_DEV */
    {
    DEV_HDR     devHdr;                 /* device header */
    BOOL       myDrvDataAvailable;      /* data is available to read */
    BOOL       myDrvRdyForWriting;      /* device is ready to write */
    SEL_WAKEUP_LIST selWakeupList;      /* list of tasks pended in select */
    } MY_DEV;
```

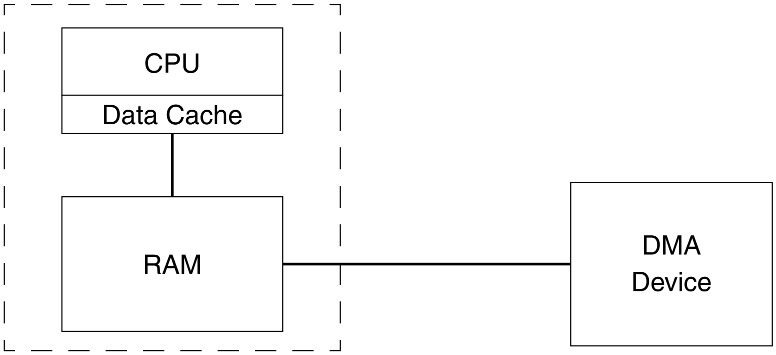
```
/* myDrv.c - code fragments for supporting select() in a driver */

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

/* First create and initialize the device */

STATUS myDrvDevCreate
    (
    char *  name,                      /* name of device to create */
    )
    {
    MY_DEV * pMyDrvDev;                /* pointer to device descriptor*/
    ... additional driver code ...

    /* allocate memory for MY_DEV */
    pMyDrvDev = (MY_DEV *) malloc (sizeof MY_DEV);
    ... additional driver code ...

    /* initialize MY_DEV */
    pMyDrvDev->myDrvDataAvailable=FALSE
    pMyDrvDev->myDrvRdyForWriting=FALSE

    /* initialize wakeup list */
    selWakeupListInit (&pMyDrvDev->selWakeupList);
    ... additional driver code ...
    }

/* ioctl function to request reading or writing */

STATUS myDrvIoctl
    (
    MY_DEV * pMyDrvDev,                /* pointer to device descriptor */
    int      request,                  /* ioctl function */
    int      arg                       /* where to send answer */
    )
    {
    ... additional driver code ...

    switch (request)
        {
        ... additional driver code ...

        case FIOSELECT:

            /* add node to wakeup list */

            selNodeAdd (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELREAD
                && pMyDrvDev->myDrvDataAvailable)
                {
                /* data available, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELWRITE
                && pMyDrvDev->myDrvRdyForWriting)
                {
                /* device ready for writing, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            break;

        case FIOUNSELECT:

            /* delete node from wakeup list */
            selNodeDelete (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);
            break;

            ... additional driver code ...
        }
    }

/* code that actually uses the select() function to read or write */

void myDrvIsr
    (
    MY_DEV * pMyDrvDev;
    )
    {
    ... additional driver code ...

    /* if there is data available to read, wake up all pending tasks */

    if (pMyDrvDev->myDrvDataAvailable)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELREAD);

    /* if the device is ready to write, wake up all pending tasks */

    if (pMyDrvDev->myDrvRdyForWriting)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELWRITE);
    }
```

Cache Coherency

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 4-7 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 4-7 **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 routines **cacheFlush( )** and **cacheInvalidate( )** are used to manually flush and invalidate cache buffers. Before a device reads the data, flush the data from the cache to RAM using **cacheFlush( )** to ensure the device reads current data. After the device has written the data into RAM, invalidate the cache entry with **cacheInvalidate( )**. This guarantees that when the data is read by the CPU, the cache is updated with the new data in RAM.

Example 4-11 **DMA Transfer Routine**

```
/* This a sample DMA transfer routine. Before programming the device
 * to output the data to the device, it flushes the cache by calling
 * cacheFlush(). On a read, after the device has transferred the data,
 * the cache entry must be invalidated using cacheInvalidate().
 */

#include "vxWorks.h"
#include "cacheLib.h"
#include "fcntl.h"
#include "example.h"
void exampleDmaTransfer   /* 1 = READ, 0 = WRITE */
    (
    UINT8 *pExampleBuf,
    int exampleBufLen,
    int xferDirection
    )
    {
    if (xferDirection == 1)
        {
        myDevToBuf (pExampleBuf);
        cacheInvalidate (DATA_CACHE, pExampleBuf, exampleBufLen);
        }

    else
        {
        cacheFlush (DATA_CACHE, pExampleBuf, exampleBufLen);
        myBufToDev (pExampleBuf);
        }
    }
```

It is possible to make a driver more efficient by combining cache-safe buffer allocation and cache-entry flushing or invalidation. The idea is to flush or invalidate a cache entry only when absolutely necessary. To address issues of cache coherency for static buffers, use **cacheDmaMalloc( )**. This routine initializes a **CACHE_FUNCS** structure (defined in **cacheLib.h**) to point to flush and invalidate routines that can be used to keep the cache coherent.

The macros **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE** use this structure to optimize the calling of the flush and invalidate routines. If the corresponding function pointer in the **CACHE_FUNCS** structure is NULL, no unnecessary flush/invalidate routines are called because it is assumed that the buffer is cache coherent (hence it is not necessary to flush/invalidate the cache entry manually).

The driver code uses a virtual address and the device uses a physical address. Whenever a device is given an address, it must be a physical address. Whenever the driver accesses the memory, it must use the virtual address.

The device driver should use **CACHE_DMA_VIRT_TO_PHYS** to translate a virtual address to a physical address before passing it to the device. It may also use

CACHE_DMA_PHYS_TO_VIRT to translate a physical address to a virtual one, but this process is time-consuming and non-deterministic, and should be avoided whenever possible.

Example 4-12 **Address-Translation Driver**

```
/* The following code is an example of a driver that performs address
 * translations. It attempts to allocate a cache-safe buffer, fill it, and
 * then write it out to the device. It uses CACHE_DMA_FLUSH to make sure
 * the data is current. The driver then reads in new data and uses
 * CACHE_DMA_INVALIDATE to guarantee cache coherency.
 */

#include "vxWorks.h"
#include "cacheLib.h"
#include "myExample.h"
STATUS myDmaExample (void)
    {
    void * pMyBuf;
    void * pPhysAddr;

    /* allocate cache safe buffers if possible */
    if ((pMyBuf = cacheDmaMalloc (MY_BUF_SIZE)) == NULL)
    return (ERROR);

    … fill buffer with useful information …

    /* flush cache entry before data is written to device */
    CACHE_DMA_FLUSH (pMyBuf, MY_BUF_SIZE);

    /* convert virtual address to physical */
    pPhysAddr = CACHE_DMA_VIRT_TO_PHYS (pMyBuf);

    /* program device to read data from RAM */
    myBufToDev (pPhysAddr);
    … wait for DMA to complete …
    … ready to read new data …

    /* program device to write data to RAM */
    myDevToBuf (pPhysAddr);
    … wait for transfer to complete …

    /* convert physical to virtual address */
    pMyBuf = CACHE_DMA_PHYS_TO_VIRT (pPhysAddr);

    /* invalidate buffer */
    CACHE_DMA_INVALIDATE (pMyBuf, MY_BUF_SIZE);
    … use data …

    /* when done free memory */
    if (cacheDmaFree (pMyBuf) == ERROR)
        return (ERROR);
    return (OK);
    }
```

4.9.4 Block Devices

General Implementation

In VxWorks, block devices have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, block device drivers interact with a file system. The file system, in turn, interacts with the I/O system. Direct access block devices have been supported since SCSI-1 and are used compatibly with dosFs and rawFs. In addition, VxWorks supports SCSI-2 sequential devices, which are organized so individual blocks of data are read and written sequentially. When data blocks are written, they are added sequentially at the end of the written medium; that is, data blocks cannot be replaced in the middle of the medium. However, data blocks can be accessed individually for reading throughout the medium. This process of accessing data on a sequential medium differs from that of other block devices.

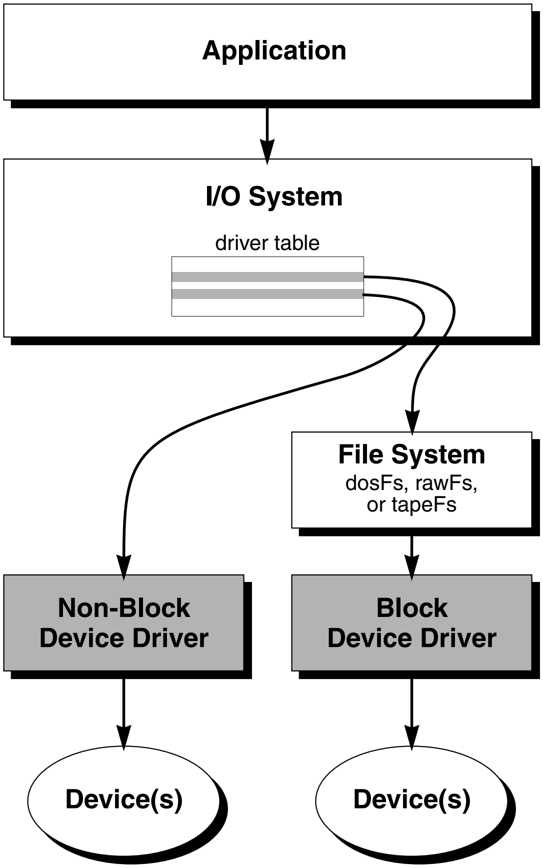
Figure 4-8 shows a layered model of I/O for both block and non-block (character) devices. This layered arrangement allows the same block device driver to be used with different file systems, and reduces the number of I/O functions that must be supported in the driver.

A device driver for a block device must provide a means for creating a logical block device structure, a **BLK_DEV** for direct access block devices or a **SEQ_DEV** for sequential block devices. The **BLK_DEV**/**SEQ_DEV** structure describes the device in a generic fashion, specifying only those common characteristics that must be known to a file system being used with the device. Fields within the structures specify various physical configuration variables for the device—for example, block size, or total number of blocks. Other fields in the structures specify routines within the device driver that are to be used for manipulating the device (reading blocks, writing blocks, doing I/O control functions, resetting the device, and checking device status). The **BLK_DEV**/**SEQ_DEV** structures also contain fields used by the driver to indicate certain conditions (for example, a disk change) to the file system.

When the driver creates the block device, the device has no name or file system associated with it. These are assigned during the device initialization routine for the chosen file system (for example, **dosFsDevInit( )** or **tapeFsDevInit( )**).

The low-level device driver for a block device is not installed in the I/O system driver table, unlike non-block device drivers. Instead, each file system in the VxWorks system is installed in the driver table as a "driver." Each file system has only one entry in the table, even though several different low-level device drivers can have devices served by that file system.

Figure 4-8 **Non-Block Devices vs. Block Devices**



After a device is initialized for use with a particular file system, all I/O operations for the device are routed through that file system. To perform specific device operations, the file system in turn calls the routines in the specified **BLK_DEV** or **SEQ_DEV** structure.

A driver for a block device must provide the interface between the device and VxWorks. There is a specific set of functions required by VxWorks; individual devices vary based on what additional functions must be provided. The user manual for the device being used, as well as any other drivers for the device, is invaluable in creating the VxWorks driver.

The following sections describe the components necessary to build low-level block device drivers that adhere to the standard interface for VxWorks file systems.

Low-Level Driver Initialization Routine

The driver normally requires a general initialization routine. This routine performs all operations that are done one time only, as opposed to operations that must be performed for each device served by the driver. As a general guideline, the operations in the initialization routine affect the whole device controller, while later operations affect only specific devices.

Common operations in block device driver initialization routines include:

- initializing hardware
- allocating and initializing data structures
- creating semaphores
- initializing interrupt vectors
- enabling interrupts

The operations performed in the initialization routine are entirely specific to the device (controller) being used; VxWorks has no requirements for a driver initialization routine.

Unlike non-block device drivers, the driver initialization routine does not call **iosDrvInstall( )** to install the driver in the I/O system driver table. Instead, the file system installs itself as a "driver" and routes calls to the actual driver using the routine addresses placed in the block device structure, **BLK_DEV** or **SEQ_DEV** (see *Device Creation Routine*, p. 178).

Device Creation Routine

The driver must provide a routine to create (define) a logical disk or sequential device. A logical disk device may be only a portion of a larger physical device. If this is the case, the device driver must keep track of any block offset values or other means of identifying the physical area corresponding to the logical device. VxWorks file systems always use block numbers beginning with zero for the start of a device. A sequential access device can be either of variable block size or fixed block size. Most applications use devices of fixed block size.

The device creation routine generally allocates a device descriptor structure that the driver uses to manage the device. The first item in this device descriptor must be a VxWorks block device structure (**BLK_DEV** or **SEQ_DEV**). It must appear first because its address is passed by the file system during calls to the driver; having the **BLK_DEV** or **SEQ_DEV** as the first item permits also using this address to identify the device descriptor.

The device creation routine must initialize the fields within the **BLK_DEV** or **SEQ_DEV** structure. The **BLK_DEV** fields and their initialization values are shown in Table 4-15.

Table 4-15 **Fields in the BLK_DEV Structure**

Field	Value
bd_blkRd	Address of the driver routine that reads blocks from the device.
bd_blkWrt	Address of the driver routine that writes blocks to the device.
bd_ioctl	Address of the driver routine that performs device I/O control.
bd_reset	Address of the driver routine that resets the device (NULL if none).
bd_statusChk	Address of the driver routine that checks disk status (NULL if none).
bd_removable	Value specifying whether or not the device is removable. TRUE if the device is removable (for example, a floppy disk); otherwise FALSE.
bd_nBlocks	Total number of blocks on the device.
bd_bytesPerBlk	Number of bytes per block on the device.
bd_blksPerTrack	Number of blocks per track on the device.
bd_nHeads	Number of heads (surfaces).
bd_retry	Number of times to retry failed reads or writes.
bd_mode	Device mode (write-protect status); generally set to **O_RDWR**.
bd_readyChanged	Value specifying whether or not the device ready status has changed. TRUE if the status has changed; initialize to TRUE to cause the disk to be mounted.

The **SEQ_DEV** fields and their initialization values are shown in Table 4-16.

The device creation routine returns the address of the **BLK_DEV** or **SEQ_DEV** structure. This address is then passed during the file system device initialization call to identify the device.

Table 4-16 **Fields in the SEQ_DEV Structure**

Field	Value
sd_seqRd	Address of the driver routine that reads blocks from the device.
sd_seqWrt	Address of the driver routine that writes blocks to the device.
sd_ioctl	Address of the driver routine that performs device I/O control.
sd_seqWrtFileMarks	Address of the driver routine that writes file marks to the device.
sd_rewind	Address of the driver routine that rewinds the sequential device.
sd_reserve	Address of the driver routine that reserves a sequential device.
sd_release	Address of the driver routine that releases a sequential device.
sd_readBlkLim	Address of the driver routine that reads the data block limits from the sequential device.
sd_load	Address of the driver routine that either loads or unloads a sequential device.
sd_space	Address of the driver routine that moves (spaces) the medium forward or backward to end-of-file or end-of-record markers.
sd_erase	Address of the driver routine that erases a sequential device.
sd_reset	Address of the driver routine that resets the device (NULL if none).
sd_statusChk	Address of the driver routine that checks sequential device status (NULL if none).
sd_blkSize	Block size of sequential blocks for the device. A block size of 0 means that variable block sizes are used.
sd_mode	Device mode (write protect status).
sd_readyChanged	Value for specifying whether or not the device ready status has changed. TRUE if the status has changed; initialize to TRUE to cause the sequential device to be mounted.
sd_maxVarBlockLimit	Maximum block size for a variable block.
sd_density	Density of sequential access media.

Unlike non-block device drivers, the device creation routine for a block device does not call **iosDevAdd( )** to install the device in the I/O system device table. Instead, this is done by the file system's device initialization routine.

Read Routine (Direct-Access Devices)

The driver must supply a routine to read one or more blocks from the device. For a direct access device, the read-blocks routine must have the following arguments and result:

```
STATUS xxBlkRd
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int       startBlk,   /* starting block to read */
    int       numBlks,    /* number of blocks to read */
    char *    pBuf        /* pointer to buffer to receive data */
    )
```

NOTE: In this and following examples, the routine names begin with *xx*. These names are for illustration only, and do not have to be used by your device driver. VxWorks references the routines by address only; the name can be anything.

pDev a pointer to the driver's device descriptor structure, represented here by the symbolic name **DEVICE**. (Actually, the file system passes the address of the corresponding **BLK_DEV** structure; these are equivalent, because the **BLK_DEV** is the first item in the device descriptor.) This identifies the device.

startBlk the starting block number to be read from the device. The file system always uses block numbers beginning with zero for the start of the device. Any offset value used for this logical device must be added in by the driver.

numBlks the number of blocks to be read. If the underlying device hardware does not support multiple-block reads, the driver routine must do the necessary looping to emulate this ability.

pBuf the address where data read from the disk is to be copied.

The read routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

Read Routine (Sequential Devices)

The driver must supply a routine to read a specified number of bytes from the device. The bytes being read are always assumed to be read from the current location of the read/write head on the media. The read routine must have the following arguments and result:

```
STATUS xxSeqRd
    (
    DEVICE *  pDev,         /* pointer to device descriptor */
    int       numBytes,     /* number of bytes to read */
    char *    buffer,       /* pointer to buffer to receive data */
    BOOL      fixed         /* TRUE => fixed block size */
    )
```

pDev a pointer to the driver's device descriptor structure, represented here by the symbolic name **DEVICE**. (Actually, the file system passes the address of the corresponding **SEQ_DEV** structure; these are equivalent, because the **SEQ_DEV** structure is the first item in the device descriptor.) This identifies the device.

numBytes the number of bytes to be read.

buffer the buffer into which *numBytes* of data are read.

fixed specifies whether the read routine reads fixed-size blocks from the sequential device or variable-sized blocks, as specified by the file system. If *fixed* is TRUE, fixed-size blocks are used.

The read routine returns **OK** if the transfer is completed successfully, or **ERROR** if a problem occurs.

Write Routine (Direct-Access Devices)

The driver must supply a routine to write one or more blocks to the device. The definition of this routine closely parallels that of the read routine. For direct-access devices, the write routine is as follows:

```
STATUS xxBlkWrt
    (
    DEVICE *  pDev,     /* pointer to device descriptor */
    int       startBlk, /* starting block for write */
    int       numBlks,  /* number of blocks to write */
    char *    pBuf      /* ptr to buffer of data to write */
    )
```

pDev a pointer to the driver's device descriptor structure.

startBlk the starting block number to be written to the device.

numBlks the number of blocks to be written. If the underlying device hardware does not support multiple-block writes, the driver routine must do the necessary looping to emulate this ability.

pBuf the address of the data to be written to the disk.

The write routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

Write Routine (Sequential Devices)

The driver must supply a routine to write a specified number of bytes to the device. The bytes being written are always assumed to be written to the current location of the read/write head on the media. For sequential devices, the write routine is as follows:

```
STATUS xxWrtTape
    (
    DEVICE *  pDev,        /* ptr to SCSI sequential device info */
    int       numBytes,    /* total bytes or blocks to be written */
    char *    buffer,      /* ptr to input data buffer      */
    BOOL      fixed        /* TRUE => fixed block size */
    )
```

pDev a pointer to the driver's device descriptor structure.

numBytes the number of bytes to be written.

buffer the buffer from which *numBytes* of data are written.

fixed specifies whether the write routine reads fixed-size blocks from the sequential device or variable-sized blocks, as specified by the file system. If *fixed* is TRUE, fixed-size blocks are used.

The write routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

I/O Control Routine

The driver must provide a routine that can handle I/O control requests. In VxWorks, most I/O operations beyond basic file handling are implemented through **ioctl( )** functions. The majority of these are handled directly by the file system. However, if the file system does not recognize a request, that request is passed to the driver's I/O control routine.

Define the driver's I/O control routine as follows:

```
STATUS xxIoctl
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int        funcCode,  /* ioctl() function code */
    int        arg        /* function-specific argument */
    )
```

pDev a pointer to the driver's device descriptor structure.

funcCode the requested **ioctl( )** function. Standard VxWorks I/O control functions are defined in the include file **ioLib.h**. Other user-defined function code values can be used as required by your device driver. The I/O control functions supported by dosFs, rawFs, and tapeFs are summarized in *5. Local File Systems* in this manual.

arg specific to the particular **ioctl( )** function requested. Not all **ioctl( )** functions use this argument.

The driver's I/O control routine typically takes the form of a multi-way switch statement, based on the function code. The driver's I/O control routine must supply a default case for function code requests it does not recognize. For such requests, the I/O control routine sets **errno** to **S_ioLib_UNKNOWN_REQUEST** and returns **ERROR**.

The driver's I/O control routine returns **OK** if it handled the request successfully; otherwise, it returns **ERROR**.

Device-Reset Routine

The driver usually supplies a routine to reset a specific device, but it is not required. This routine is called when a VxWorks file system first mounts a disk or tape, and again during retry operations when a read or write fails.

Declare the driver's device-reset routine as follows:

```
STATUS xxReset
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

When called, this routine resets the device and controller. Do not reset other devices, if it can be avoided. The routine returns **OK** if the driver succeeded in resetting the device; otherwise, it returns **ERROR**.

If no reset operation is required for the device, this routine can be omitted. In this case, the device-creation routine sets the *xx*_**reset** field in the **BLK_DEV** or **SEQ_DEV** structure to NULL.

NOTE: In this and following examples, the names of fields in the **BLK_DEV** and **SEQ_DEV** structures are parallel except for the initial letters **bd_** or **sd_**. In these cases, the initial letters are represented by *xx_*, as in the *xx*_**reset** field to represent both the **bd_reset** field and the **sd_reset** field.

Status-Check Routine

If the driver provides a routine to check device status or perform other preliminary operations, the file system calls this routine at the beginning of each **open( )** or **creat( )** on the device.

Define the status-check routine as follows:

```
STATUS xxStatusChk
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

The routine returns **OK** if the open or create operation can continue. If it detects a problem with the device, it sets **errno** to some value indicating the problem, and returns **ERROR**. If **ERROR** is returned, the file system does not continue the operation.

A primary use of the status-check routine is to check for a disk change on devices that do not detect the change until after a new disk is inserted. If the routine determines that a new disk is present, it sets the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE and returns **OK** so that the open or create operation can continue. The new disk is then mounted automatically by the file system. (See *Change in Ready Status*, p.186.)

Similarly, the status check routine can be used to check for a tape change. This routine determines whether a new tape has been inserted. If a new tape is present, the routine sets the **sd_readyChanged** field in the **SEQ_DEV** structure to TRUE and returns **OK** so that the open or create operation can continue. The device driver should not be able to unload a tape, nor should a tape be physically ejected, while a file descriptor is open on the tape device.

If the device driver requires no status-check routine, the device-creation routine sets the *xx*_**statusChk** field in the **BLK_DEV** or **SEQ_DEV** structure to NULL.

Write-Protected Media

The device driver may detect that the disk or tape in place is write-protected. If this is the case, the driver sets the *xx*_**mode** field in the **BLK_DEV** or **SEQ_DEV** structure to **O_RDONLY**. This can be done at any time (even after the device is initialized for use with the file system). The file system respects the *xx*_**mode** field setting and does not allow writes to the device until the *xx*_**mode** field is changed to **O_RDWR** or **O_WRONLY**.

Change in Ready Status

The driver informs the file system whenever a change in the device's ready status is recognized. This can be the changing of a floppy disk, changing of the tape medium, or any other situation that makes it advisable for the file system to remount the disk.

To announce a change in ready status, the driver sets the *xx*_**readyChanged** field in the **BLK_DEV** or **SEQ_DEV** structure to TRUE. This is recognized by the file system, which remounts the disk during the next I/O initiated on the disk. The file system then sets the *xx*_**readyChanged** field to FALSE. The *xx*_**readyChanged** field is never cleared by the device driver.

Setting *xx*_**readyChanged** to TRUE has the same effect as calling the file system's ready-change routine (for example, calling **ioctl( )** with the **FIODISKCHANGE** function code).

An optional status-check routine (see *Status-Check Routine*, p.185) can provide a convenient mechanism for asserting a ready-change, particularly for devices that cannot detect a disk change until after the new disk is inserted. If the status-check routine detects that a new disk is present, it sets *xx*_**readyChanged** to TRUE. This routine is called by the file system at the beginning of each open or create operation.

Write-File-Marks Routine (Sequential Devices)

The sequential driver must provide a routine that can write file marks onto the tape device. The write file marks routine must have the following arguments:

```
STATUS xxWrtFileMarks
    (
    DEVICE *  pDev,          /* pointer to device descriptor */
    int       numMarks,      /* number of file marks to write */
    BOOL      shortMark      /* short or long file marks */
    )
```

pDev a pointer to the driver's device descriptor structure.

numMarks the number of file marks to be written sequentially.

shortMark the type of file mark (short or long). If *shortMark is* TRUE, short marks are written.

The write file marks routine returns **OK** if the file marks are written correctly on the tape device; otherwise, it returns **ERROR**.

Rewind Routine (Sequential Devices)

The sequential driver must provide a rewind routine in order to rewind tapes in the tape device. The rewind routine is defined as follows:

```
STATUS xxRewind
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

When called, this routine rewinds the tape in the tape device. The routine returns **OK** if completion is successful; otherwise, it returns **ERROR**.

Reserve Routine (Sequential Devices)

The sequential driver can provide a reserve routine that reserves the physical tape device for exclusive access by the host that is executing the reserve routine. The tape device remains reserved until it is released by that host, using a release routine, or by some external stimulus.

The reserve routine is defined as follows:

```
STATUS xxReserve
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

If a tape device is reserved successfully, the reserve routine returns **OK**. However, if the tape device cannot be reserved or an error occurs, it returns **ERROR**.

Release Routine (Sequential Devices)

This routine releases the exclusive access that a host has on a tape device. The tape device is then free to be reserved again by the same host or some other host. This routine is the opposite of the reserve routine and must be provided by the driver if the reserve routine is provided.

The release routine is defined as follows:

```
STATUS xxReset
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

If the tape device is released successfully, this routine returns **OK**. However, if the tape device cannot be released or an error occurs, this routine returns **ERROR**.

Read-Block-Limits Routine (Sequential Devices)

The read-block-limits routine can poll a tape device for its physical block limits. These block limits are then passed back to the file system so the file system can decide the range of block sizes to be provided to a user.

The read-block-limits routine is defined as follows:

```
STATUS xxReadBlkLim
    (
    DEVICE *  pDev,            /* pointer to device descriptor */
    int      *maxBlkLimit,     /* maximum block size for device */
    int      *minBlkLimit      /* minimum block size for device */
    )
```

pDev a pointer to the driver's device descriptor structure.

maxBlkLimit
returns the maximum block size that the tape device can handle to the calling tape file system.

minBlkLimit
returns the minimum block size that the tape device can handle.

The routine returns **OK** if no error occurred while acquiring the block limits; otherwise, it returns **ERROR**.

Load/Unload Routine (Sequential Devices)

The sequential device driver must provide a load or unload routine in order to mount or unmount tape volumes from a physical tape device. Loading means that a volume is being mounted by the file system. This is usually done with an **open( )** or a **creat( )** call. However, a device should be unloaded or unmounted only when the file system wants to eject the tape volume from the tape device.

The load/unload routine is defined as follows:

```
STATUS xxLoad
    (
    DEVICE *  pDev,  /* pointer to device descriptor */
    BOOL      load   /* load or unload device */
    )
```

pDev a pointer to the driver's device descriptor structure.

load a boolean variable that determines if the tape is loaded or unloaded. If *load* is TRUE, the tape is loaded. If *load* is FALSE, the tape is unloaded.

The load/unload routine returns **OK** if the load or unload operation ends successfully; otherwise, it returns **ERROR**.

Space Routine (Sequential Devices)

The sequential device driver must provide a space routine that moves, or spaces, the tape medium forward or backward. The amount of distance that the tape spaces depends on the kind of search that must be performed. In general, tapes can be searched by end-of-record marks, end-of-file marks, or other types of device-specific markers.

The basic definition of the space routine is as follows; however, other arguments can be added to the definition:

```
STATUS xxSpace
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int       count,      /* number of spaces */
    int       spaceCode   /* type of space */
    )
```

pDev a pointer to the driver's device descriptor structure.

count specifies the direction of search. A positive *count* value represents forward movement of the tape device from its current location (forward space); a negative *count* value represents a reverse movement (back space).

spaceCode defines the type of space mark that the tape device searches for on the tape medium. The basic types of space marks are end-of-record and end-of-file. However, different tape devices may support more sophisticated kinds of space marks designed for more efficient maneuvering of the medium by the tape device.

If the device is able to space in the specified direction by the specified count and space code, the routine returns **OK**; if these conditions cannot be met, it returns **ERROR**.

Erase Routine (Sequential Devices)

The sequential driver must provide a routine that allows a tape to be erased. The erase routine is defined as follows:

```
STATUS xxErase
    (
    DEVICE *  pDev  /* pointer to driver's device descriptor structure */
    )
```

The routine returns **OK** if the tape is erased; otherwise, it returns **ERROR**.

4.9.5 Driver Support Libraries

The subroutine libraries in Table 4-17 may assist in the writing of device drivers. Using these libraries, drivers for most devices that follow standard protocols can be written with only a few pages of device-dependent code. See the reference entry for each library for details.

Table 4-17 **VxWorks Driver Support Routines**

Library	Description
errnoLib	Error status library
ftpLib	ARPA File Transfer Protocol library
ioLib	I/O interface library
iosLib	I/O system library
intLib	Interrupt support subroutine library
remLib	Remote command library
rngLib	Ring buffer subroutine library
ttyDrv	Terminal driver
wdLib	Watchdog timer subroutine library

4.10 PCMCIA

A PCMCIA card can be plugged into notebook computers to connect devices such as modems and external hard drives.[2] VxWorks provides PCMCIA facilities for **pcPentium, pcPentium2**, **pcPentium3** and 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, add the **INCLUDE_PCMCIA** component to the VxWorks kernel protection domain. For information about PCMCIA facilities, see the entries for **pcmciaLib** and **pcmciaShow** in the *VxWorks API Reference*.

[2] PCMCIA stands for Personal Computer Memory Card International Association, and refers to both the association and the standards that it has developed.

4.11 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 entries for **pciAutoConfigLib**, **pciConfigLib**, **pciInitLib**, and **pciConfigShow** in the *VxWorks API Reference*.

5
Local File Systems

5.1 Introduction

VxWorks uses a standard I/O interface between the file system and the device driver. This allows multiple file systems, of the same or different types, to operate within a single VxWorks system. By following these standard interfaces, you can write your own file system for VxWorks, and freely mix file systems and device drivers.

This chapter discusses the VxWorks file systems, listed below, describing how they are organized, configured, and used.

- **dosFs.** Designed for real-time use of block devices (disks) and compatible with the MS-DOS file system.
- **rawFS.** Provides a simple *raw file system* that essentially treats an entire disk as a single large file.
- **tapeFs.** Designed for tape devices that do not use a standard file or directory structure on tape. Essentially treats the tape volume as a raw device in which the entire volume is a large file.
- **cdromFs.** Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system.
- **TSFS (Target Server File System) .** Uses the Tornado target server to provide the target with access to files on the host system.

VxWorks also provides support for flash memory devices through the optional product TrueFFS. For more information, see *8. Flash Memory Block Device Driver*.

5.2 MS-DOS-Compatible File System: dosFs

The dosFs file system is an MS-DOS-compatible file system that offers considerable flexibility appropriate to the multiple demands of real-time applications. The primary features are:

- Hierarchical files and directories, allowing efficient organization and an arbitrary number of files to be created on a volume.
- A choice of contiguous or non-contiguous files on a per-file basis.
- Compatibility with widely available storage and retrieval media (diskettes, hard drives, and so on).
- The ability to boot VxWorks from a dosFs file system.
- Support for VFAT (Microsoft VFAT long file names) and VXLONGS (VxWorks long file names) directory formats.
- Support for FAT12, FAT16, and FAT32 file allocation table types.

For API reference information about dosFs, see the entries for **dosFsLib** and **dosFsFmtLib**, as well as the **cbioLib**, **dcacheCbio**, and **dpartCbio** entries, in the *VxWorks API Reference*.

For information about the MS-DOS file system, please see the Microsoft documentation.

NOTE: The discussion in this chapter of the dosFs file system uses the term *sector* to refer to the minimum addressable unit on a *disk*. This definition of the term follows most MS-DOS documentation. However, in VxWorks, these units on the disk are normally referred to as *blocks*, and a disk device is called a *block device*.

5.2.1 Creating a dosFs File System

This section summarizes the process of creating a dosFs file system, outlining the steps involved and diagramming where this process fits into the VxWorks system.

The process described in these steps corresponds to building the layers of VxWorks components between the hardware (such as a SCSI disk) and the I/O system, as illustrated in the sections within the dotted line in Figure 5-1.

Step 1: Configure the Kernel

Configure your kernel with the dosFs, CBIO, and block device components. This step is described in *5.2.2 Configuring Your System*, p.197.

Step 2: Initialize the dosFs File System

This step is done automatically if you have included the required components in your project. This step is described in *5.2.3 Initializing the dosFs File System*, p.198

Step 3: Create the Block Device

Create either a block device or a CBIO driver device (**ramDiskCbio**). This step is described in *5.2.4 Creating a Block Device*, p.198.

Step 4: Create a Disk Cache

Creating a disk cache is optional. Disk cache is intended only for rotational media. This step is described in *5.2.5 Creating a Disk Cache*, p.198.

Step 5: Create the Partition for Use

Creating and mounting partitions is optional. This step is described in *5.2.6 Creating and Using Partitions*, p.198.

Step 6: Create a dosFs Device

Create the dosFs device. You can safely create the device whether or not you are using a pre-formatted disk. This step is described in *5.2.7 Creating a dosFs Device*, p.201.

Step 7: Format the Volume

If you are not using pre-formatted disks, format the volumes. This step is described in *5.2.8 Formatting the Volume*, p.201.

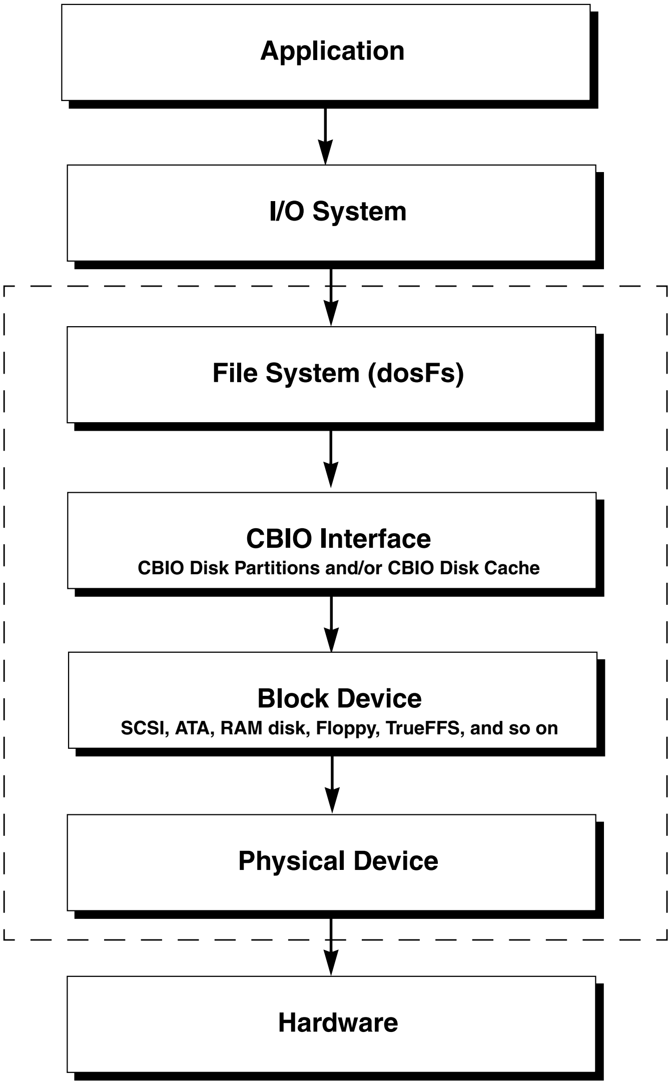
Step 8: Check the Disk Volume Integrity

Optionally, check the disk for volume integrity using **dosFsChkDsk( )**. Disk checking large disks can be time-consuming. The parameters you pass to **dosFsDevCreate( )** determine whether disk checking happens automatically. For details, see the entry for **dosFsDevCreate( )** in the *VxWorks Reference Manual*.

Step 9: Mount the Disk

A disk volume is *mounted* automatically, generally during the first **open( )** or **creat( )** operation for a file or directory on the disk. This step is described in *5.2.9 Mounting Volumes*, p.203.

Figure 5-1 **Configuring a VxWorks System with dosFs**


Application
I/O System
File System (dosFs)
CBIO Interface
CBIO Disk Partitions and/or CBIO Disk Cache
Block Device
SCSI, ATA, RAM disk, Floppy, TrueFFS, and so on
Physical Device
Hardware

5.2.2 Configuring Your System

To include dosFs in your VxWorks-based system, configure the kernel with the appropriate components for the dosFs file system.

Required Components

The following components are required:

- **INCLUDE_DOSFS_MAIN.** **dosFsLib** (2)
- **INCLUDE_DOSFS_FAT.** dosFs FAT12/16/32 FAT handler
- **INCLUDE_CBIO.** CBIO API module

And, either one or both of the following components are required:

- **INCLUDE_DOSFS_DIR_VFAT.** Microsoft VFAT direct handler
- **INCLUDE_DOSFS_DIR_FIXED.** Strict 8.3 & VxLongNames directory handler

In addition, you need to include the appropriate component for your block device; for example, **INCLUDE_SCSI** or **INCLUDE_ATA**. Finally, add any related components that are required for your particular system.

Optional dosFs Components

Optional dosFs components are:

- **INCLUDE_DOSFS.** u**srDosFsOld.c** wrapper layer
- **INCLUDE_DOSFS_FMT.** dosFs2 file system formatting module
- **INCLUDE_DOSFS_CHKDSK.** file system integrity checking
- **INCLUDE_DISK_UTIL.** standard file system operations, such as **ls**, **cd**, **mkdir**, **xcopy**, and so on
- **INCLUDE_TAR.** the **tar** utility

Optional CBIO Components

Optional CBIO components are:

- **INCLUDE_DISK_CACHE.** CBIO API disk caching layer
- **INCLUDE_DISK_PART.** disk partition handling code
- **INCLUDE_RAM_DISK.** CBIO API RAM disk driver

5.2.3 Initializing the dosFs File System

Before any other operations can be performed, the dosFs file system library, **dosFsLib**, must be initialized. This happens automatically, triggered by the required dosFs components that were included in the system.

Initializing the file system invokes **iosDrvInstall( )**, which adds the driver to the I/O system driver table. The driver number assigned to the dosFs file system is recorded in a global variable, **dosFsDrvNum**. The table specifies the entry points for the dosFs file operations that are accessed by the devices using dosFs.

5.2.4 Creating a Block Device

Next, create one or more block devices. To create the device, call the routine appropriate for that device driver. The format for this routine is *xxx***DevCreate( )** where *xxx* represents the device driver type; for example, **scsiBlkDevCreate( )** or **ataDevCreate( )**.

The driver routine returns a pointer to a block device descriptor structure, **BLK_DEV**. This structure describes the physical attributes of the device and specifies the routines that the device driver provides to a file system. For more information on block devices, see *4.9.4 Block Devices*, p. 176.

5.2.5 Creating a Disk Cache

If you have included the **INCLUDE_DISK_CACHE** component in your system, you can use **dcacheDevCreate( )** to create a disk cache for each block device. Disk cache is intended to reduce the impact of **seek** times on rotational media, and is not used for RAM disks or TrueFFS devices. Example 5-1 creates a disk cache.

5.2.6 Creating and Using Partitions

If you have included the **INCLUDE_DISK_PART** component in your system, you can create partitions on a disk and mount volumes atop the partitions. Use the **usrFdiskPartCreate( )** and **dpartDevCreate( )** routines to do this.

The following two examples create and use partitions. The first example creates, partitions, and formats a disk. The second example uses the partitioned disk, from the first example, to create the partition handler.

Example 5-1 **Creating and Partitioning a Disk and Creating Volumes**

This example takes a pointer to a block device, creates three partitions, creates the partition handler for these partitions, and creates the dosFs device handler for them. Then, it formats the partitions using **dosFsVolFormat( )**, which is discussed in the next section.

```
STATUS usrPartDiskFsInit
    (
    void * blkDevId               /* CBIO_DEV_ID or BLK_DEV*   */
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    int dcacheSize = 0x30000 ;
    CBIO_DEV_ID cbio, cbio1 ;
    /* create disk cache */

    if((cbio = dcacheDevCreate(blkDevId, NULL, dcacheSize, "/sd0"))
        == NULL )
        return ERROR ;

    /* create partitions */

    if((usrFdiskPartCreate (cbio,3,50,45)) == ERROR)
        return ERROR;

    /* create partition manager with FDISK style decoder, up to 3 parts */

    if((cbio1 = dpartDevCreate( cbio, 3, usrFdiskPartRead )) == NULL)
        return ERROR;

    /* create the 1st file system, 8 simult. open files, with CHKDSK */

    if(dosFsDevCreate( devNames[0], dpartPartGet(cbio1,0), 8, 0 ) == ERROR)
        return ERROR;

    /* create the 2nd file system, 6 simult. open files, with CHKDSK */

    if(dosFsDevCreate( devNames[1], dpartPartGet(cbio1,1), 6, 0 ) == ERROR)
        return ERROR;

    /* create the 3rd file system, 4 simultaneously open files, no CHKDSK */

    if(dosFsDevCreate( devNames[2], dpartPartGet(cbio1,2), 4, NONE )
       == ERROR)
        return ERROR;

    /* Formatting the first partition */

    if(dosFsVolFormat (devNames[0], 2,0) == ERROR)
        return ERROR;

    /* Formatting the second partition */
    if(dosFsVolFormat (devNames[1], 2,0) == ERROR)
        return ERROR;

    /* Formatting the third partition */

    if(dosFsVolFormat (devNames[2], 2,0) == ERROR)
        return ERROR;

    return OK;
    }
```

Example 5-2 **Accessing a Partitioned Disk**

The following example configures a partitioned disk with three already existing partitions. Note that the ATA hard disk component allows for auto-mounting as many partitions as are referenced within its name parameter.

```
STATUS usrPartDiskFsInit
    (
    void * blkDevId              /* CBIO_DEV_ID or BLK_DEV*/
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    int dcacheSize = 0x30000 ;
    CBIO_DEV_ID cbio, cbio1 ;

    /* create disk cache */

    if((cbio = dcacheDevCreate(blkDevId, NULL, dcacheSize, "/sd0"))
       == NULL )
        return ERROR ;

    /* create partition manager with FDISK style decoder, up to 3 parts */

    if((cbio1 = dpartDevCreate( cbio, 3, usrFdiskPartRead )) == NULL)
        return ERROR;

    /* create the 1st file system, 8 simultaneously open files
     * with CHKDSK
     */

    if(dosFsDevCreate( devNames[0], dpartPartGet(cbio1,0), 8, 0 )
       == ERROR)
        return ERROR;

    /* create the 2nd file sys, 6 simultaneously open files, with CHKDSK */

    if(dosFsDevCreate( devNames[1], dpartPartGet(cbio1,1), 6, 0 ) == ERROR)
        return ERROR;

    /* create the 3rd file system, 4 simultaneously open files, no CHKDSK */

    if(dosFsDevCreate( devNames[2], dpartPartGet(cbio1,2), 4, 0)
       ==ERROR)
```

```
        return ERROR;

    return OK;
    }
```

5.2.7 Creating a dosFs Device

Create a dosFs device using **dosFsDevCreate( )**, which calls **iosDrvAdd( )** internally. This step simply adds the device to the I/O system; however, it does not invoke any I/O operations and, therefore, does not mount the disk.

This disk is not mounted until the first I/O operation occurs. For more information, see *5.4.4 Mounting Volumes*, p.225.

5.2.8 Formatting the Volume

If you are using an unformatted disk, format the volume in either of two ways:

- By calling **dosFsVolFormat( )** directly, specifying options for both the format of the FAT and the directory format (described below).
- By issuing the **ioctl( ) FIODISKINIT** command, which invokes the formatting routine with **dosFsLib**. This method uses the default volume format and parameters.

For more details, see the *VxWorks API Reference* entries for **dosFsVolFormat( )** and **ioctl( )**.

The MS-DOS and dosFs file systems provide options for the format of the File Allocation Table (FAT) and the format of the directory. These options, described below, are completely independent.

⚠ **CAUTION:** If you are using a disk that is already initialized with an MS-DOS boot sector, FAT, and root directory—for example, by using the **FORMAT** utility in MS-DOS—you can use **dosFsDevCreate( )** to create a dosFs device. However, *do not* call **dosFsVolFormat( )** or the file system data structures will be re-initialized (reformatted).

File Allocation Table (FAT) Formats

A volume FAT format is set during disk formatting, according to either the volume size (by default), or the per-user defined settings passed to **dosFsVolFormat( )**. FAT options are summarized in Table 5-1:

Table 5-1 **FAT Formats**

Format	FAT Table Entry SIze	Usage	Size
FAT12	12 bits per cluster number	Appropriate for very small devices with up to 4,084 KB clusters.	Typically, each cluster is two sectors large.
FAT16	16 bits per cluster number	Appropriate for small disks of up to 65,524 KB clusters.	Typically, used for volumes up to 2 GB; can support up to 8 GB.
FAT32	32 bits (only 28 used) per cluster number	Appropriate for medium and larger disk drives.	By convention, used for volumes larger than 2 GB.

Directory Formats

There are three options for the directory format. These are:

- **MSFT Long Names (VFAT)** Uses case-insensitive long filenames, with up to 254 characters. This format accepts disks created with short names. MSFT Long Names[1] is the default directory format.

- **Short Names (8.3)** Case-insensitive MS-DOS-style filenames (8.3), with eight uppercase characters for the *name* itself and three for the *extension*.

- **VxWorks Long Names (VxLong)** Wind River's proprietary VxWorks long name support, introduced prior to MSFT Long Names and used for backward compatibility with old dosFs VxLong disks. Allows case-sensitive filenames of up to 40 characters (consisting of any ASCII characters). The dot character (**.**), which indicates a file-name extension in MS-DOS, has no special significance in dosFs VxLong.

NOTE: The VxWorks long names format supports 40-bit file size fields, allowing the file size to be *larger* than 4 GB.

[1] The MSFT Long Names (VFAT) format supports 32-bit file size fields, limiting the file size to a 4 GB maximum.

WARNING: If you use VxWorks Long Names, the disk will not be MS-DOS compatible. Use this long name support only for storing data local to VxWorks, on a disk that is initialized on a VxWorks system.

5.2.9 Mounting Volumes

A disk volume is *mounted* automatically, generally during the first **open( )** or **creat( )** operation for a file or directory on the disk. Certain **ioctl( )** calls also cause the disk to be mounted.

NOTE: The dosFs initialization procedure succeeds even if a volume is unformatted or a removable diskette is not inserted in the drive at the time the system boots. A system can boot successfully and initialize all its devices even if a drive has no removable media in it and the media's configuration and parameters are unknown.

5.2.10 Demonstrating with Examples

This section provides examples of the steps discussed in the sections above. These examples use a variety of configurations and device types. They are meant to be relatively generic and applicable to most block devices.

The first example uses an ATA disk, and includes detailed descriptions of the commands run from the shell, displaying both user input and command-line output. The second example lists the required steps to create and format a RAM disk volume. The last example demonstrates how to initialize a pre-formatted SCSI disk.

CAUTION: Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name; unexpected results may occur.

Example 5-3 **Initializing an ATA Disk with dosFs2**

This example demonstrates how to initialize an ATA disk with dosFs2. This example displays the commands and output from the VxWorks shell. While these steps use an ATA block device type, they are applicable to other block devices.

Step 1: Create the Block Device

Create a block device (**BLK_DEV**) that controls the master ATA hard disk (drive zero) on the primary ATA controller (controller zero). This block device uses the entire disk.

```
-> pAta = ataDevCreate (0,0,0,0)
new symbol "pAta" added to symbol table.
pAta = 0x3fff334: value = 67105604 = 0x3fff344 = pAta + 0x10
```

Above, **pAta** is now a block device pointer (**BLK_DEV** *). The routine **ataDevCreate( )** returns a valid value. A return value of **NULL** (0x0) indicates an error in **ataDevCreate( )**. Such an error usually indicates a BSP configuration or hardware configuration error.

This step is appropriate for any **BLK_DEV** device; for example, flash, SCSI, and so on. For related information, see the reference entry for **ataDevCreate( )**.

Step 2: Create a Disk Cache

Next, create an (optional) disk data cache CBIO layer atop this disc:

```
-> pCbioCache = dcacheDevCreate (pAta,0,0,"ATA Hard Disk Cache")
new symbol "pCbioCache" added to symbol table.
pCbioCache = 0x3ffdbd0: value = 67105240 = 0x3fff1d8
```

Above, **pCbioCache** is a **CBIO_DEV_ID**, which is the handle for controlling the CBIO disk cache layer. For more information, see the reference entry for **dcacheDevCreate( )**.

Step 3: Create Partitions

Then, create two partitions on this disk device, specifying 50% of the disk space for the second partition, leaving 50% for the first partition. This step should only be performed once, when the disk is first initialized. If partitions are already written to the disk, this step should not be performed since it destroys data:

```
-> usrFdiskPartCreate (pCbioCache, 2, 50, 0, 0)
value = 0 = 0x0
```

For more information, see the entry for **usrFdiskPartLibCbio( )** in the *VxWorks Reference Manual*.

In this step, the block device pointer **pAta** could have been passed instead of **pCbioCache**. Doing so would cause **usrFdiskPartCreate( )** to use the **BLK_DEV** routines via a wrapper internally created by **cbioWrapBlkDev( )**.

Step 4: Display Partitions

Now, you can optionally display the partitions created with **usrFdiskPartShow( )**:

```
-> usrFdiskPartShow (pAta)

Master Boot Record - Partition Table
--------------------------------------
Partition Entry number 00     Partition Entry offset 0x1be
Status field = 0x80           Primary (bootable) Partition
Type 0x06: MSDOS 16-bit FAT >=32M Partition
Partition start LCHS: Cylinder 0000, Head 001, Sector 01
Partition end   LCHS: Cylinder 0245, Head 017, Sector 39
Sectors offset from MBR partition 0x00000027
Number of sectors in partition 0x00262c17
Sectors offset from start of disk 0x00000027

Master Boot Record - Partition Table
--------------------------------------
Partition Entry number 01     Partition Entry offset 0x1ce
Status field = 0x00           Non-bootable Partition
Type 0x06: MSDOS 16-bit FAT >=32M Partition
Partition start LCHS: Cylinder 0000, Head 018, Sector 01
Partition end   LCHS: Cylinder 0233, Head 067, Sector 39
Sectors offset from MBR partition 0x00262c3e
Number of sectors in partition 0x00261d9e
Sectors offset from start of disk 0x00262c3e

Master Boot Record - Partition Table
--------------------------------------
Partition Entry number 02     Partition Entry offset 0x1de
Status field = 0x00           Non-bootable Partition
Type 0x00: Empty (NULL) Partition

Master Boot Record - Partition Table
--------------------------------------
Partition Entry number 03     Partition Entry offset 0x1ee
Status field = 0x00           Non-bootable Partition
Type 0x00: Empty (NULL) Partition

value = 0 = 0x0
->
```

Note above that two partitions have been created upon the disc, and the remaining two partition table entries are blank.

NOTE: The CBIO device ID **pCbioCache** could have been passed to **usrFdiskPartShow( )**, instead of the block device pointer **pAta**. Doing so would cause **usrFdiskPartShow( )** to use the CBIO disk cache routines to access the device.

Step 5: Create a Partition Handler

Next, create a partition handler/mounter upon this disk. When using a disk cache layer, the partition handler code should always be instantiated after the disk cache layer:

```
-> pCbioParts = dpartDevCreate (pCbioCache,2, usrFdiskPartRead)
new symbol "pCbioParts" added to symbol table.
pCbioParts = 0x3ffd92c: value = 67099276 = 0x3ffda8c = pCbioParts + 0x160
->
```

The call to **dpartDevCreate( )** informs the partition layer to expect two partitions on the disk (24 is the maximum number of partitions the mounter can handle.) We have also instructed the partition manager (**dpartCbio**) to use the **FDISK** style partition mounter code **usrFdiskPartRead( )**.

For more information, see the reference entry for **dpartDevCreate( )**.

Step 6: Create the dosFs2 File System

Create dosFs2 file systems atop each partition. The last argument specifies the integrated **chkdsk** configuration.

```
-> dosFsDevCreate ("/DOSA", dpartPartGet (pCbioParts,0), 16, 0)
value = 0 = 0x0
-> dosFsDevCreate ("/DOSB", dpartPartGet (pCbioParts,1), 16, -1)
value = 0 = 0x0
->

-> devs
drv name
0 /null
1 /tyCo/0
1 /tyCo/1
6 ahostname:
3 /DOSA       <---- First Partition
3 /DOSB       <---- Second Partition
value = 25 = 0x19
->
```

This step defines the volume parameters and adds them to the IO system; it does not format the volumes. No disk I/O is performed during this step. The volumes are not mounted at this stage.

For more information, see the entry for **dosFsLib** in the *VxWorks Reference Manual*.

Step 7: Format the DOS Volumes

Next, format the DOS volumes. This step need only be done once, when the volumes are first initialized. If the DOS volumes have already been initialized (formatted), then omit this step. The example formats the file system volumes with default options:

```
-> dosFsVolFormat ("/DOSA",0,0)
Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT16, sectors per cluster 32
2 FAT copies, 0 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId (null)  , serial number 8120000
Label:"           " ...
Disk with 2501655 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 64
2 FAT copies, 39082 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId VXDOS16 , serial number 8120000
Label:"           " ...
value = 0 = 0x0

-> dosFsVolFormat ("/DOSB",0,0)
Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT16, sectors per cluster 32
2 FAT copies, 0 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId (null) , serial number 9560000
Label:"           " ...

Disk with 2497950 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 64
2 FAT copies, 39024 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId VXDOS16 , serial number 9560000
Label:"           " ...
value = 0 = 0x0
->
```

For more information, see the entry for **dosFsFmtLib** in the *VxWorks Reference Manual*.

Step 8: Access the Volumes

Now, the dosFs volumes are ready to access. Note that **/DOSA** will start a default **chkdsk** code and that **/DOSB** will not. **Autochk** is set via the fourth argument to **dosFsDevCreate( )**.

```
-> ll "/DOSA"
/DOSA/  - disk check in progress ...
/DOSA/  - Volume is OK
```

```
total # of clusters:        39,085
# of free clusters:         39,083
# of bad clusters:          0
total free space:           1,221 Mb
max contiguous free space:  1,280,671,744 bytes
# of files:                 0
# of folders:               0
total bytes in files:       0
# of lost chains:           0
total bytes in lost chains: 0

Listing Directory /DOSA:
value = 0 = 0x0

-> ll "/DOSB"

Listing Directory /DOSB:
value = 0 = 0x0
->

-> dosFsShow "/DOSB"

volume descriptor ptr (pVolDesc):       0x3f367c0
cache block I/O descriptor ptr (cbio):  0x3f37be0
auto disk check on mount:               NOT ENABLED
max # of simultaneously open files:     18
file descriptors in use:                0
# of different files in use:            0
# of descriptors for deleted files:     0
# of  obsolete descriptors:             0

current volume configuration:
- volume label:                NO LABEL ; (in boot sector: )
- volume Id:                   0x9560000
- total number of sectors:     2,497,950
- bytes per sector:            512
- # of sectors per cluster:    64
- # of reserved sectors:       1
- FAT entry size:              FAT16
- # of sectors per FAT copy:   153
- # of FAT table copies:       2
- # of hidden sectors:         39
- first cluster is in sector # 339
- Update last access date for open-read-close = FALSE
- directory structure:         VFAT
- root dir start sector:       307
- # of sectors per root:       32
- max # of entries in root:    512

FAT handler information:
------------------------
- allocation group size:       4 clusters
- free space on volume:        1,278,771,200 bytes
value = 0 = 0x0
->
```

Above, we can see the **Volume** parameters for the **/DOSB** volume. The file system volumes are now mounted and ready to be exercised.

If you are working with an ATA hard disk or a CD-ROM file system from an ATAPI CD-ROM drive, you can, alternatively, use **usrAtaConfig( )**. This routine processes several steps at once. For more information, see the reference entry.

Example 5-4 **Creating and Formatting a RAM Disk Volume**

The following example creates a RAM disk of a certain size, and formats it for use with the dosFs file system. This example uses the **ramDiskCbio** module, which is intended for direct use with **dosFsLib**:

```
STATUS usrRamDiskInit
    (
    void                              /* no argument */
    )
    {
    int ramDiskSize = 128 * 1024 ;    /* 128KB, 128 bytes per sector */
    char *ramDiskDevName = "/ram0" ;
    CBIO_DEV_ID cbio ;

    /* 128 bytes/sec, 17 secs/track, auto-allocate */

    cbio = ramDiskDevCreate(NULL, 128, 17, ramDiskSize/128, 0) ;

    if( cbio == NULL )
        return ERROR ;

    /* create the file system, 4 simultaneously open files, no CHKDSK */

    dosFsDevCreate( ramDiskDevName, cbio, 4, NONE );

    /* format the RAM disk, ignore memory contents */

    dosFsVolFormat( cbio, DOS_OPT_BLANK | DOS_OPT_QUIET, NULL );

    return OK;
    }
```

Example 5-5 **Initializing a SCSI Disk Drive**

This example initializes a SCSI disk as a single file system volume (and assumes that the disk is already formatted).

```
STATUS usrScsiDiskInit
    (
    int scsiId                        /* SCSI id */
    )
    {
```

```
int dcacheSize = 128 * 1024 ;     /* 128KB disk cache */
char *diskDevName = "/sd0" ;      /* disk device name */
CBIO_DEV_ID cbio;                 /* pointer to a CBIO_DEV */
BLK_DEV *pBlk;                    /* pointer to a BLK_DEV */
SCSI_PHYS_DEV *pPhys ;            /* pointer to a SCSI physical device */

/* Create the SCSI physical device */

if ((pPhys = scsiPhysDevCreate
    (pSysScsiCtrl, scsiId, 0, 0, NONE, 0,0, 0))==  NULL)
    {
    printErr ("usrScsiDiskInit: scsiPhysDevCreate SCSI ID %d failed.\n",
              scsiId, 0, 0, 0, 0, 0);
    return ERROR;
    }

/* Create the block device */

if( (pblk = scsiBlkDevCreate(pPhys, 0, NONE )) == NULL )
    return ERROR;

/*
 * works for ids less than 10
 * append SCSI id to make the device name unique
 */

diskDevName[strlen(diskDevName)-1] += scsiId ;

/* create disk cache */

if((cbio = dcacheDevCreate(pblk, NULL, dcacheSize, diskDevName))
   == NULL )
    return ERROR ;

/* create the file system, 10 simultaneously open files, with CHKDSK */

dosFsDevCreate( diskDevName, cbio, 10, 0 );

return OK;
}
```

5.2.11 *Working with Volumes and Disks*

This section discusses issues related to disks and volumes.

For more information about **ioctl( )** support functions, see *5.2.16 I/O Control Functions Supported by dosFsLib*, p.219.

Announcing Disk Changes with Ready-Change

You can inform **dosFsLib** that a disk change is taking place by using the *ready-change* mechanism. A change in the disk's ready-status is interpreted by **dosFsLib** as indicating that the disk must be remounted before the next I/O operation. To announce a ready-change, use any of the following methods:

- Call **ioctl( )** with the **FIODISKCHANGE** function.
- Have the device driver set the **bd_readyChanged** field in the **BLK_DEV** structure to **TRUE**; this has the same effect as notifying **dosFsLib** directly.
- Use **cbioRdyChgdSet( )** to set the ready-changed bit in the CBIO layer.

Accessing Volume Configuration Information

The **dosFsShow( )** routine can be used to display volume configuration information. The **dosFsVolDescGet( )** routine will programmatically obtain or verify a pointer to the **DOS_VOLUME_DESC** structure. For more information, see the reference entries.

Synchronizing Volumes

When a disk is *synchronized*, all modified buffered data is physically written to the disk, so that the disk is up to date. This includes data written to files, updated directory information, and the FAT. To avoid loss of data, a disk should be synchronized before it is removed. For more information, see the entries for **close( )** and **dosFsVolUnmount( )** in the *VxWorks Reference Manual*.

5.2.12 Working with Directories

This section discusses issues related to directories.

Creating Subdirectories

For FAT32, subdirectories can be created in any directory at any time. For FAT12 and FAT16, subdirectories can be created in any directory at any time, except in the root directory once it reaches its maximum entry count. Subdirectories can be created in the following ways:

1. Using **ioctl( )** with the **FIOMKDIR** function: The name of the directory to be created is passed as a parameter to **ioctl( )**.
2. Using **open( )**: To create a directory, the **O_CREAT** option must be set in the *flags* parameter to open, and the **FSTAT_DIR** option must be set in the *mode* parameter. The **open( )** call returns a file descriptor that describes the new directory. Use this file descriptor for reading only and close it when it is no longer needed.
3. Use **mkdir( )**, **usrFsLib**.

When creating a directory using any of the above methods, the new directory name must be specified. This name can be either a full pathname or a pathname relative to the current working directory.

Removing Subdirectories

A directory that is to be deleted must be empty (except for the "**.**" and "**..**" entries). The root directory can never be deleted. Subdirectories can be removed in the following ways:

1. Using **ioctl( )** with the **FIORMDIR** function, specifying the name of the directory. Again, the file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
2. Using the **remove( )** function, specifying the name of the directory.
3. Use **rmdir( )**, **usrFsLib**.

Reading Directory Entries

You can programmatically search directories on dosFs volumes using the **opendir( )**, **readdir( )**, **rewinddir( )**, and **closedir( )** routines.

To obtain more detailed information about a specific file, use the **fstat( )** or **stat( )** routine. Along with standard file information, the structure used by these routines also returns the file-attribute byte from a directory entry.

For more information, see the entry for **dirLib** in the VxWorks API Reference.

5.2.13 Working with Files

This section discusses issues related to files.

File I/O

Files on a dosFs file system device are created, deleted, written, and read using the standard VxWorks I/O routines: **creat( )**, **remove( )**, **write( )**, and **read( )**. For more information, see *4.3 Basic I/O*, p. 111, and the **ioLib** entries in the *VxWorks API Reference*.

File Attributes

The file-attribute byte in a dosFs directory entry consists of a set of flag bits, each indicating a particular file characteristic. The characteristics described by the file-attribute byte are shown in Table 5-2.

Table 5-2 **Flags in the File-Attribute Byte**

VxWorks Flag Name	Hex Value	Description
DOS_ATTR_RDONLY	**0x01**	read-only file
DOS_ATTR_HIDDEN	**0x02**	hidden file
DOS_ATTR_SYSTEM	**0x04**	system file
DOS_ATTR_VOL_LABEL	**0x08**	volume label
DOS_ATTR_DIRECTORY	**0x10**	subdirectory
DOS_ATTR_ARCHIVE	**0x20**	file is subject to archiving

DOS_ATTR_RDONLY
If this flag is set, files accessed with **open( )** cannot be written to. If the **O_WRONLY** or **O_RDWR** flags are set, **open( )** returns **ERROR**, setting **errno** to **S_dosFsLib_READ_ONLY**.

DOS_ATTR_HIDDEN
This flag is ignored by **dosFsLib** and produces no special handling. For example, entries with this flag are reported when searching directories.

DOS_ATTR_SYSTEM
This flag is ignored by **dosFsLib** and produces no special handling. For example, entries with this flag are reported when searching directories.

DOS_ATTR_VOL_LABEL
This is a volume label flag, which indicates that a directory entry contains the dosFs volume label for the disk. A label is not required. If used, there can be only one volume label entry per volume, in the root directory. The volume label entry is not reported when reading the contents of a directory (using **readdir( )**). It can only be determined using the **ioctl( )** function **FIOLABELGET**. The volume label can be set (or reset) to any string of 11 or fewer characters, using the **ioctl( )** function **FIOLABELSET**. Any file descriptor open to the volume can be used during these **ioctl( )** calls.

DOS_ATTR_DIRECTORY
This is a directory flag, which indicates that this entry is a subdirectory, and not a regular file.

DOS_ATTR_ARCHIVE
This is an archive flag, which is set when a file is created or modified. This flag is intended for use by other programs that search a volume for modified files and selectively archive them. Such a program must clear the archive flag, since VxWorks does not.

All the flags in the attribute byte, except the directory and volume label flags, can be set or cleared using the **ioctl( )** function **FIOATTRIBSET**. This function is called after the opening of the specific file with the attributes to be changed. The attribute-byte value specified in the **FIOATTRIBSET** call is copied directly; to preserve existing flag settings, determine the current attributes using **stat( )** or **fstat( )**, then change them using bitwise AND and OR operations.

Example 5-6 **Setting DosFs File Attributes**

This example makes a dosFs file read-only, and leaves other attributes intact.

```
STATUS changeAttributes
    (
    void
    )
    {
    int           fd;
    struct stat   statStruct;

    /* open file */

    if ((fd = open ("file", O_RDONLY, 0)) == ERROR)
        return (ERROR);
    /* get directory entry data */

    if (fstat (fd, &statStruct) == ERROR)
        return (ERROR);

    /* set read-only flag on file */

    if (ioctl (fd, FIOATTRIBSET, (statStruct.st_attrib | DOS_ATTR_RDONLY))
        == ERROR)
        return (ERROR);

    /* close file */

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

NOTE: You can also use the **attrib( )** routine to change file attributes. For more information, see the entry in **usrFsLib**.

5.2.14 Disk Space Allocation Options

The dosFs file system allocates disk space using one of the following methods. The first two methods are selected based upon the size of the write operation. The last method must be manually specified.

- **single cluster allocation.** *Single cluster allocation* uses a single cluster, which is the minimum allocation unit. This method is automatically used when the write operation is smaller than the size of a single cluster.
- **cluster group allocation (nearly contiguous).** *Cluster group allocation* uses adjacent (contiguous) groups of clusters, called *extents*. Cluster group allocation is nearly contiguous allocation and is the default method used when files are written in units larger than the size of a disk's cluster.
- **absolutely contiguous allocation .** *Absolutely contiguous allocation* uses only absolutely contiguous clusters. Because this type of allocation is dependent upon the existence of such space, it is specified under only two conditions: immediately after a new file is created and when reading from a file assumed to have been allocated to a contiguous space. Using this method risks disk fragmentation.

For any allocation method, you can deallocate unused reserved bytes by using the POSIX-compatible routine **ftruncate( )** or the **ioctl( )** function **FIOTRUNC**.

Choosing an Allocation Method

Under most circumstances, cluster group allocation is preferred to absolutely contiguous file access. Because it is nearly contiguous file access, it achieves a nearly optimal access speed. Cluster group allocation also significantly minimizes the risk of fragmentation posed by absolutely contiguous allocation.

Absolutely contiguous allocation attains raw disk throughput levels, however this speed is only slightly faster than nearly contiguous file access. Moreover, fragmentation is likely to occur over time. This is because after a disk has been in use for some period of time, it becomes impossible to allocate contiguous space. Thus, there is no guarantee that new data, appended to a file created or opened with absolutely continuous allocation, will be contiguous to the initially written data segment.

It is recommended that for a performance-sensitive operation, the application regulate disk space utilization, limiting it to 90% of the total disk space. Fragmentation is unavoidable when filling in the last free space on a disk, which has a serious impact on performance.

Using Cluster Group Allocation

The dosFs file system defines the size of a cluster group based on the media's physical characteristics. That size is fixed for each particular media. Since seek operations are an overhead that reduces performance, it is desirable to arrange files so that sequential portions of a file are located in physically contiguous disk clusters. Cluster group allocation occurs when the cluster group size is considered sufficiently large so that the seek time is negligible compared to the **read**/**write** time. This technique is sometimes referred to as "*nearly contiguous*" *file access* because seek time between consecutive cluster groups is significantly reduced.

Because all large files on a volume are expected to have been written as a group of extents, removing them frees a number of extents to be used for new files subsequently created. Therefore, as long as free space is available for subsequent file storage, there are always extents available for use. Thus, cluster group allocation effectively prevents *fragmentation* (where a file is allocated in small units spread across distant locations on the disk). Access to fragmented files can be extremely slow, depending upon the degree of fragmentation.

Using Absolutely Contiguous Allocation

A contiguous file is made up of a series of consecutive disk sectors. Absolutely contiguous allocation is intended to allocate contiguous space to a specified file (or directory) and, by so doing, optimize access to that file. You can specify absolutely contiguous allocation either when creating a file, or when opening a file previously created in this manner.

For more information on the **ioctl( )** functions, see *5.2.16 I/O Control Functions Supported by dosFsLib*, p.219.

Allocating Contiguous Space for a File

To allocate a contiguous area to a newly created file, follow these steps:

1. First, create the file in the normal fashion using **open( )** or **creat( )**.
2. Then, call **ioctl( )**. Use the file descriptor returned from **open( )** or **creat( )** as the file descriptor argument. Specify **FIOCONTIG** as the function code argument and the size of the requested contiguous area, in bytes, as the third argument.

The FAT is then searched for a suitable section of the disk. If found, this space is assigned to the new file. The file can then be closed, or it can be used for further I/O operations. The file descriptor used for calling **ioctl( )** should be the only descriptor open to the file. Always perform the **ioctl( ) FIOCONTIG** operation before writing any data to the file.

To request the largest available contiguous space, use **CONTIG_MAX** for the size of the contiguous area. For example:

```
status = ioctl (fd, FIOCONTIG, CONTIG_MAX);
```

Allocating Space for Subdirectories

Subdirectories can also be allocated a contiguous disk area in the same manner:

- If the directory is created using the **ioctl( )** function **FIOMKDIR**, it must be subsequently opened to obtain a file descriptor to it.
- If the directory is created using options to **open( )**, the returned file descriptor from that call can be used.

A directory must be empty (except for the “**.**” and “**..**” entries) when it has contiguous space allocated to it.

Opening and Using a Contiguous File

Fragmented files require following cluster chains in the FAT. However, if a file is recognized as contiguous, the system can use an enhanced method that improves performance. This applies to all contiguous files, whether or not they were explicitly created using **FIOCONTIG**. Whenever a file is opened, it is checked for contiguity. If it is found to be contiguous, the file system registers the necessary information about that file to avoid the need for subsequent access to the FAT table. This enhances performance when working with the file by eliminating seek operations.

When you are opening a contiguous file, you can explicitly indicate that the file is contiguous by specifying the **DOS_O_CONTIG_CHK** flag with **open( )**. This prompts the file system to retrieve the section of contiguous space, allocated for this file, from the FAT table.

Demonstrating with an Example

To find the maximum contiguous area on a device, you can use the **ioctl( )** function **FIONCONTIG**. This information can also be displayed by **dosFsConfigShow( )**.

Example 5-7 **Finding the Maximum Contiguous Area on a DosFs Device**

In this example, the size (in bytes) of the largest contiguous area is copied to the integer pointed to by the third parameter to **ioctl( )** (*count*).

```
STATUS contigTest
    (
    void                    /* no argument */
    )
    {
    int count;              /* size of maximum contiguous area in bytes */
    int fd;                 /* file descriptor */

    /* open device in raw mode */

    if ((fd = open ("/DEV1/", O_RDONLY, 0)) == ERROR)
        return (ERROR);

    /* find max contiguous area */

    ioctl (fd, FIONCONTIG, &count);

    /* close device and display size of largest contiguous area */

    close (fd);
    printf ("largest contiguous area = %d\n", count);
    return (OK);
    }
```

5.2.15 Crash Recovery and Volume Consistency

The DOS file system is inherently susceptible to data structure inconsistencies that result from interruptions during certain types of disk updates. These types of interruptions include power failures, inadvertent system crashes for fixed disks, and the manual removal of a disk.

NOTE: The DOS file system is not considered a fault-tolerant file system.

The inconsistencies occur because the file system data for a single file is stored in three separate regions of the disk. The data stored in these regions are:

- The file chain in the File Allocation Table (FAT), located in a region near the beginning of the disk.
- The directory entry, located in a region anywhere on the disk.
- File clusters containing file data, located anywhere on the disk.

Since all three regions cannot be always updated before an interruption, dosFs includes an optional integrated consistency-checking mechanism to detect and recover from inconsistencies. For example, if a disk is removed when a file is being deleted, a consistency check completes the file deletion operation. Or, if a file is being created when an interruption occurs, then the file is un-created. In other words, the consistency checker either rolls forward or rolls back the operation that has the inconsistency, making whichever correction is possible.

To use consistency checking, specify the *autoChkLevel* parameter to **dosFsDevCreate( )**, invoke it manually, or call the **chkdsk( )** utility. If configured, consistency checking is invoked under the following conditions:

- When a new volume is mounted.
- Once at system initialization time for fixed disks.
- Every time a new cartridge is inserted for removable disks.

NOTE: Consistency checking slows a system down, particularly when a a disk is first accessed.

5.2.16 I/O Control Functions Supported by dosFsLib

The dosFs file system supports the **ioctl( )** functions. These functions are defined in the header file **ioLib.h** along with their associated constants.

For more information, see the manual entries for **dosFsLib** and for **ioctl( )** in **ioLib.**

Table 5-3 **I/O Control Functions Supported by dosFsLib**

Function	Decimal Value	Description
FIOATTRIBSET	35	Sets the file-attribute byte in the dosFs directory entry.
FIOCONTIG	36	Allocates contiguous disk space for a file or directory.
FIODISKCHANGE	13	Announces a media change.
FIODISKFORMAT	5	Formats the disk (device driver function).
FIODISKINIT	6	Initializes a dosFs file system on a disk volume.
FIOFLUSH	2	Flushes the file output buffer.
FIOFSTATGET	38	Gets file status information (directory entry data).
FIOGETNAME	18	Gets the filename of the *fd*.
FIOLABELGET	33	Gets the volume label.
FIOLABELSET	34	Sets the volume label.
FIOMKDIR	31	Creates a new directory.
FIOMOVE	47	Moves a file (does not rename the file).
FIONCONTIG	41	Gets the size of the maximum contiguous area on a device.
FIONFREE	30	Gets the number of free bytes on the volume.
FIONREAD	1	Gets the number of unread bytes in a file.
FIOREADDIR	37	Reads the next directory entry.
FIORENAME	10	Renames a file or directory.
FIORMDIR	32	Removes a directory.
FIOSEEK	7	Sets the current byte offset in a file.
FIOSYNC	21	Same as **FIOFLUSH**, but also re-reads buffered file data.
FIOTRUNC	42	Truncates a file to a specified length.
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position in a file.

5.3 Booting from a Local dosFs File System Using SCSI

VxWorks can be booted from a local SCSI device. Before you can boot from SCSI, you must make a new boot application ROM that contains the SCSI library. Include **INCLUDE_SCSI**, **INCLUDE_SCSI_BOOT**, and **SYS_SCSI_CONFIG** in your boot application project.

After burning the SCSI boot application ROM, you can prepare the dosFs file system for use as a boot device. The simplest way to do this is to partition the SCSI device so that a dosFs file system starts at block 0. You can then make the new system image, place it on your SCSI boot device, and boot the new VxWorks system. These steps are shown in more detail below.

⚠ **WARNING:** For use as a boot device, the directory name for the dosFs file system must begin and end with slashes (as with **/sd0/** used in the following example). This is an exception to the usual naming convention for dosFs file systems and is incompatible with the NFS requirement that device names not end in a slash.

Step 1: Create the SCSI Device

Create the SCSI device using **scsiPhysDevCreate( )** (see *SCSI Drivers*, p. 146), and initialize the disk with a dosFs file system (see *5.2.3 Initializing the dosFs File System*, p. 198). Modify the file *installDir*/**target/***bspName*/**sysScsi.c** to reflect your SCSI configuration.

Step 2: Rebuild Your System

Rebuild your system.

Step 3: Copy the VxWorks Runtime Image

Copy the file **vxWorks** to the drive. Below, a VxWorks task spawns the **copy( )** routine, passing it two arguments.

The first argument is the source file for the **copy( )** command. The source file is the VxWorks runtime image, **vxWorks**. The source host name is **tiamat**:, the source filename is **C:/vxWorks**. These are passed to **copy( )** in concatenated form, as the string "**tiamat:C:/vxWorks**."

The second argument is the destination file for the **copy( )** command. The dosFs file system, on the local target SCSI disk device, is named **/sd0**, and the target file name is **vxWorks**. These are, similarly, passed to **copy( )** in concatenated form, as the string "**/sd0/vxWorks**." When booting the target from the SCSI device, the bootrom image should specify the runtime file as "**/sd0/vxWorks**".

```
-> sp (copy, "tiamat:c:/vxWorks","/sd0/vxWorks")
task spawned: id = 0x3f2a200, name = t2
value = 66232832 = 0x3f2a200

Copy OK: 1065570 bytes copied
```

Step 4: Copy the System Symbol Table

Depending upon image configuration, the **vxWorks.sym** file for the system symbol table may also be needed. Therefore, in similar fashion, copy the **vxWorks.sym** file. The runtime image, **vxWorks**, downloads the **vxWorks.sym** file from the same location.

```
-> sp (copy, "tiamat:c:/vxWorks.sym","/sd0/vxWorks.sym")
task spawned: id = 0x3f2a1bc, name = t3
value = 66232764 = 0x3f2a1bc

Copy OK: 147698 bytes copied
```

Step 5: Test the Copying

Now, list the files to ensure that the files were correctly copied.

```
-> sp (ll, "/sd0")
task spawned: id = 0x3f2a1a8, name = t4
value = 66232744 = 0x3f2a1a8
->

Listing Directory /sd0:
-rwxrwxrwx  1 0       0          1065570 Oct 26 2001 vxWorks
-rwxrwxrwx  1 0       0           147698 Oct 26 2001 vxWorks.sym
```

Step 6: Reboot and Change Parameters

Reboot the system, and then change the boot parameters. Boot device parameters for SCSI devices follow this format:

scsi=*id*,*lun*

where *id* is the SCSI ID of the boot device, and *lun* is its Logical Unit Number (LUN). To enable use of the network, include the on-board Ethernet device (for example, **ln** for LANCE) in the *other* field.

The following example boots from a SCSI device with a SCSI ID of 2 and a LUN of 0.

```
boot device            : scsi=2,0
processor number       : 0
host name              : host
file name              : /sd0/vxWorks
inet on ethernet (e)   : 147.11.1.222:ffffff00
host inet (h)          : 147.11.1.3
```

```
user (u)                : jane
flags (f)               : 0x0
target name (tn)        : t222
other                   : ln
```

5.4 Raw File System: rawFs

VxWorks provides a minimal “file system,” rawFs, for use in systems that require only the most basic disk I/O functions. The rawFs file system, implemented in **rawFsLib**, treats the entire disk volume much like a single large file.

Although the dosFs file system provides this ability to varying degrees, the rawFs file system offers advantages in size and performance if more complex functions are not required.

To use the rawFs file system in a VxWorks-based system, include the **INCLUDE_RAWFS** component in the kernel, and set the **NUM_RAWFS_FILES** parameter to the desired maximum open file descriptor count.

5.4.1 Disk Organization

The rawFs file system imposes no organization of the data on the disk. It maintains no directory information; thus there is no division of the disk area into specific files. All **open( )** operations on rawFs devices specify only the device name; no additional filenames are possible.

The entire disk area is treated as a single file and is available to any file descriptor that is open for the device. All read and write operations to the disk use a byte-offset relative to the start of the first block on the disk.

5.4.2 Initializing the rawFs File System

Before any other operations can be performed, the rawFs library, **rawFsLib**, must be initialized by calling **rawFsInit( )**. This routine takes a single parameter, the maximum number of rawFs file descriptors that can be open at one time. This count is used to allocate a set of descriptors; a descriptor is used each time a rawFs device is opened.

The **rawFsInit( )** routine also makes an entry for the rawFs file system in the I/O system driver table (with **iosDrvInstall( )**). This entry specifies the entry points for rawFs file operations, for all devices that use the rawFs file system. The driver number assigned to the rawFs file system is placed in a global variable, **rawFsDrvNum**.

The **rawFsInit( )** routine is normally called by the **usrRoot( )** task after starting the VxWorks system.

5.4.3 Initializing a Device for Use With rawFs

After the rawFs file system is initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine (*xx***DevCreate( )**). The driver routine returns a pointer to a block device descriptor structure (**BLK_DEV**). The **BLK_DEV** structure describes the physical aspects of the device and specifies the routines in the device driver that a file system can call. For more information on block devices, see *4.9.4 Block Devices*, p.176.

Immediately after its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with rawFs, the already-created block device must be associated with rawFs and a name must be assigned to it. This is done with the **rawFsDevInit( )** routine. Its parameters are the name to be used to identify the device and a pointer to the block device descriptor structure (**BLK_DEV**):

```
RAW_VOL_DESC *pVolDesc;
BLK_DEV      *pBlkDev;
pVolDesc = rawFsDevInit ("DEV1:", pBlkDev);
```

The **rawFsDevInit( )** call assigns the specified name to the device and enters the device in the I/O system device table (with **iosDevAdd( )**). It also allocates and initializes the file system's volume descriptor for the device. It returns a pointer to the volume descriptor to the caller; this pointer is used to identify the volume during certain file system calls.

Note that initializing the device for use with rawFs does not format the disk. That is done using an **ioctl( )** call with the **FIODISKFORMAT** function.

NOTE: No disk initialization (**FIODISKINIT**) is required, because there are no file system structures on the disk. Note, however, that rawFs accepts that **ioctl( )** function code for compatibility with other file systems; in such cases, it performs no action and always returns **OK**.

5.4.4 Mounting Volumes

A disk volume is *mounted* automatically, generally during the first **open( )** or **creat( )** operation. (Certain **ioctl( )** functions also cause the disk to be mounted.) The volume is again mounted automatically on the first disk access following a ready-change operation (see *5.4.6 Changing Disks*, p.225).

⚠ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name; unexpected results may occur.

5.4.5 File I/O

To begin I/O operations upon a rawFs device, first open the device using the standard **open( )** function. (The **creat( )** function can be used instead, although nothing is actually "created.") Data on the rawFs device is written and read using the standard I/O routines **write( )** and **read( )**. For more information, see *4.3 Basic I/O*, p.111.

The character pointer associated with a file descriptor (that is, the byte offset where the read and write operations take place) can be set by using **ioctl( )** with the **FIOSEEK** function.

Multiple file descriptors can be open simultaneously for a single device. These must be carefully managed to avoid modifying data that is also being used by another file descriptor. In most cases, such multiple open descriptors use **FIOSEEK** to set their character pointers to separate disk areas.

5.4.6 Changing Disks

The rawFs file system must be notified when removable disks are changed (for example, when floppies are swapped). Two different notification methods are provided: (1) **rawFsVolUnmount( )** and (2) the ready-change mechanism.

Un-mounting Volumes

The first method of announcing a disk change is to call **rawFsVolUnmount( )** prior to removing the disk. This call flushes all modified file descriptor buffers if possible (see *Synchronizing Volumes*, p.227) and also marks any open file descriptors as obsolete. The next I/O operation remounts the disk. Calling **ioctl( )** with **FIOUNMOUNT** is equivalent to using **rawFsVolUnmount( )**. Any open file descriptor to the device can be used in the **ioctl( )** call.

Attempts to use obsolete file descriptors for further I/O operations produce an **S_rawFsLib_FD_OBSOLETE** error. To free an obsolete descriptor, use **close( )**, as usual. This frees the descriptor even though it produces the same error.

ISRs must not call **rawFsVolUnmount( )** directly, because the call can pend while the device becomes available. The ISR can instead give a semaphore that prompts a task to un-mount the volume. (Note that **rawFsReadyChange( )** can be called directly from ISRs; see *Announcing Disk Changes with Ready-Change*, p.226.)

When **rawFsVolUnmount( )** is called, it attempts to write buffered data out to the disk. Its use is therefore inappropriate for situations where the disk-change notification does not occur until a new disk is inserted, because the old buffered data would be written to the new disk. In this case, use **rawFsReadyChange( )**, as described in *Announcing Disk Changes with Ready-Change*, p.226.

If **rawFsVolUnmount( )** is called after the disk is physically removed, the data flushing portion of its operation fails. However, the file descriptors are still marked as obsolete, and the disk is marked as requiring remounting. An error is *not* returned by **rawFsVolUnmount( )**; to avoid lost data in this situation, explicitly synchronize the disk before removing it (see *Synchronizing Volumes*, p.227).

Announcing Disk Changes with Ready-Change

The second method of announcing that a disk change is taking place is with the *ready-change* mechanism. A change in the disk's ready-status is interpreted by **rawFsLib** to indicate that the disk must be remounted during the next I/O call.

There are three ways to announce a ready-change:

- By calling **rawFsReadyChange( )** directly.
- By calling **ioctl( )** with **FIODISKCHANGE**.
- By having the device driver set the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE; this has the same effect as notifying **rawFsLib** directly.

The ready-change announcement does not cause buffered data to be flushed to the disk. It merely marks the volume as needing remounting. As a result, data written to files can be lost. This can be avoided by synchronizing the disk before asserting ready-change. The combination of synchronizing and asserting ready-change provides all the functionality of **rawFsVolUnmount( )** except for marking file descriptors as obsolete.

Ready-change can be called from an ISR, because it does not attempt to flush data or perform other operations that could cause delay.

The block device driver status-check routine (identified by the **bd_statusChk** field in the **BLK_DEV** structure) is useful for asserting ready-change for devices that only detect a disk change after the new disk is inserted. This routine is called at the beginning of each **open( )** or **creat( )**, before the file system checks for ready-change.

If it is not possible for a ready-change to be announced each time the disk is changed, close all file descriptors for the volume before changing the disk.

Synchronizing Volumes

When a disk is *synchronized*, all buffered data that is modified is written to the physical device so that the disk is up to date. For the rawFs file system, the only such data is that contained in open file descriptor buffers.

To avoid loss of data, synchronize a disk before removing it. You may need to explicitly synchronize a disk, depending on when (or if) the **rawFsVolUnmount( )** call is issued.

When **rawFsVolUnmount( )** is called, an attempt is made to synchronize the device before un-mounting. If this disk is still present and writable at the time of the call, synchronization takes place automatically; there is no need to synchronize the disk explicitly.

However, if the **rawFsVolUnmount( )** call is made after a disk is removed, it is obviously too late to synchronize, and **rawFsVolUnmount( )** discards the buffered data. Therefore, make a separate **ioctl( )** call with the **FIOSYNC** function before removing the disk. (For example, this could be done in response to an operator command.) Any open file descriptor to the device can be used during the **ioctl( )** call. This call writes all modified file descriptor buffers for the device out to the disk.

5.4.7 I/O Control Functions Supported by rawFsLib

The rawFs file system supports the **ioctl( )** functions shown in Table 5-4. The functions listed are defined in the header file **ioLib.h**. For more information, see the manual entries for **rawFsLib** and for **ioctl( )** in **ioLib**.

Table 5-4 **I/O Control Functions Supported by rawFsLib**

Function	Decimal Value	Description
FIODISKCHANGE	13	Announces a media change.
FIODISKFORMAT	5	Formats the disk (device driver function).
FIODISKINIT	6	Initializes the rawFs file system on a disk volume (optional).
FIOFLUSH	2	Same as **FIOSYNC**.
FIOGETNAME	18	Gets the device name of the *fd*.
FIONREAD	1	Gets the number of unread bytes on the device.
FIOSEEK	7	Sets the current byte offset on the device.
FIOSYNC	21	Writes out all modified file descriptor buffers.
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position on the device.

5.5 Tape File System: tapeFs

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

To configure VxWorks with tapeFs, include the **INCLUDE_TAPEFS** component in the kernel.

NOTE: The tape file system must be configured with SCSI-2 enabled. See *SCSI Drivers*, p.146 for configuration details.

5.5.1 Tape Organization

The tapeFs file system imposes no organization of the data on the tape volume. It maintains no directory information; there is no division of the tape area into specific files; and no filenames are used. An **open( )** operation on the tapeFs device specifies only the device name; no additional filenames are allowed.

The entire tape area is available to any file descriptor open for the device. All read and write operations to the tape use a location offset relative to the current location of the tape head. When a file is configured as a rewind device and first opened, tape operations begin at the beginning-of-medium (BOM); see *Initializing a Device for Use With tapeFs*, p.229. Thereafter, all operations occur relative to where the tape head is located at that instant of time. No location information, as such, is maintained by tapeFs.

5.5.2 Initializing the tapeFs File System

The tapeFs file system must be initialized, and a tape device created, in order to make the physical tape device available to the tape file system. At this point, normal I/O system operations can be performed.

The tapeFs library, **tapeFsLib**, is initialized by calling **tapeFsInit( )**. Each tape file system can handle multiple tape devices. However, each tape device is allowed only one file descriptor. Thus, you cannot open two files on the same tape device.

The **tapeFsInit( )** routine also makes an entry for the tapeFs file system in the I/O system driver table (with **iosDrvInstall( )**). This entry specifies function pointers to carry out tapeFs file operations on devices that use the tapeFs file system. The driver number assigned to the tapeFs file system is placed in a global variable, **tapeFsDrvNum**.

When initializing a tape device, **tapeFsInit( )** is called automatically if **tapeFsDevInit( )** is called; thus, the tape file system does not require explicit initialization.

Initializing a Device for Use With tapeFs

Once the tapeFs file system has been initialized, the next step is to create one or more devices that can be used with it. This is done using the sequential device creation routine, **scsiSeqDevCreate( )**. The driver routine returns a pointer to a sequential device descriptor structure, **SEQ_DEV**. The **SEQ_DEV** structure describes the physical aspects of the device and specifies the routines in the device driver that tapeFs can call. For more information on sequential devices, see the manual entry for **scsiSeqDevCreate( )**, *Configuring SCSI Drivers*, p.147 and *4.9.4 Block Devices*, p.176.

Immediately after its creation, the sequential device has neither a name nor a file system associated with it. To initialize a sequential device for use with tapeFs, call **tapeFsDevInit( )** to assign a name and declare a file system. Its parameters are the volume name—for identifying the device; a pointer to **SEQ_DEV**—the sequential device descriptor structure; and a pointer to an initialized tape configuration structure **TAPE_CONFIG.** This structure has the following form:

```
typedef struct  /* TAPE_CONFIG tape device config structure */
    {
    int blkSize;            /* block size; 0 => var. block size */
    BOOL rewind;            /* TRUE => a rewind device; FALSE => no rewind */
    int numFileMarks;       /* not used */
    int density;            /* not used */
    } TAPE_CONFIG;
```

In the preceding definition of **TAPE_CONFIG**, only two fields, **blkSize** and **rewind**, are currently in use. If **rewind** is TRUE, then a tape device is rewound to the beginning-of-medium (BOM) upon closing a file with **close( )**. However, if **rewind** is **FALSE**, then closing a file has no effect on the position of the read/write head on the tape medium.

The **blkSize** field specifies the block size of the physical tape device. Having set the block size, each read or write operation has a transfer unit of **blkSize**. Tape devices can perform fixed or variable block transfers, a distinction also captured in the **blkSize** field.

For more information on initializing a tapeFs device, see the *VxWorks API Reference* entry for **tapeFsDevInit( )**.

Systems with Fixed Block and Variable Block Devices

A tape file system can be created for fixed block size transfers or variable block size transfers, depending on the capabilities of the underlying physical device. The type of data transfer (fixed block or variable block) is usually decided when the tape device is being created in the file system, that is, before the call to **tapeFsDevInit( )**. A block size of zero represents variable block size data transfers.

Once the block size has been set for a particular tape device, it is usually not modified. To modify the block size, use the **ioctl( )** functions **FIOBLKSIZESET** and **FIOBLKSIZEGET** to set and get the block size on the physical device.

Note that for fixed block transfers, the tape file system buffers a block of data. If the block size of the physical device is changed after a file is opened, the file should first be closed and then re-opened in order for the new block size to take effect.

Example 5-8 **Tape Device Configuration**

There are many ways to configure a tape device. In this code example, a tape device is configured with a block size of 512 bytes and the option to rewind the device at the end of operations.

```
/* global variables assigned elsewhere */

SCSI_PHYS_DEV *   pScsiPhysDev;

/* local variable declarations */

TAPE_VOL_DESC *  pTapeVol;
SEQ_DEV  *       pSeqDev;
TAPE_CONFIG      tapeConfig;

/* initialization code */

tapeConfig.blkSize  = 512;
tapeConfig.rewind   = TRUE;
pSeqDev    = scsiSeqDevCreate (pScsiPhysDev);
pTapeVol   = tapeFsDevInit ("/tape1", pSeqDev, tapeConfig);
```

The **tapeFsDevInit( )** call assigns the specified name to the device and enters the device in the I/O system device table (with **iosDevAdd( )**). The return value of this routine is a pointer to a volume descriptor structure that contains volume-specific configuration and state information.

5.5.3 Mounting Volumes

A tape volume is *mounted* automatically during the **open( )** operation. There is no specific mount operation, that is, the mount is implicit in the **open( )** operation.

⚠ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name; unexpected results may occur.

The tapeFs tape volumes can be operated in only one of two modes: read-only (**O_RDONLY**) or write-only (**O_WRONLY**). There is no read-write mode. The mode of operation is defined when the file is opened using **open( )**.

5.5.4 File I/O

To begin I/O to a tapeFs device, the device is first opened using **open( )**. Data on the tapeFs device is written and read using the standard I/O routines **write( )** and **read( )**. For more information, see *4.7.7 Block Devices*, p.145.

End-of-file markers can be written using **ioctl( )** with the **MTWEOF** function. For more information, see *5.5.6 I/O Control Functions Supported by tapeFsLib*, p.232.

5.5.5 Changing Tapes

The tapeFs file system should be notified when removable media are changed (for example, when tapes are swapped). The **tapeFsVolUnmount( )** routine controls the mechanism to un-mount a tape volume.

A tape should be un-mounted before it is removed. Prior to un-mounting a tape volume, an open file descriptor must be closed. Closing an open file flushes any buffered data to the tape, thus synchronizing the file system with the data on the tape. To flush or synchronize data before closing the file, call **ioctl( )** with the **FIOFLUSH** or **FIOSYNC** functions.

After closing any open file, call **tapeFsVolUnmount( )** before removing the tape. Once a tape has been un-mounted, the next I/O operation must remount the tape using **open( )**.

Interrupt handlers must not call **tapeFsVolUnmount( )** directly, because it is possible for the call to pend while the device becomes available. The interrupt handler can instead give a semaphore that prompts a task to un-mount the volume.

5.5.6 I/O Control Functions Supported by tapeFsLib

The tapeFs file system supports the **ioctl( )** functions shown in Table 5-5. The functions listed are defined in the header files **ioLib.h**, **seqIo.h**, and **tapeFsLib.h.** For more information, see the *VxWorks API Reference* entries for **tapeFsLib**, **ioLib**, and **ioctl( )**.

Table 5-5 **I/O Control Functions Supported by tapeFsLib**

Function	Decimal Value	Description
FIOFLUSH	2	Writes out all modified file descriptor buffers.
FIOSYNC	21	Same as **FIOFLUSH**.
FIOBLKSIZEGET	1001	Gets the actual block size of the tape device by issuing a driver command to it. Check this value with that set in the **SEQ_DEV** data structure.
FIOBLKSIZESET	1000	Sets the block size of the tape device on the device and in the **SEQ_DEV** data structure.
MTIOCTOP	1005	Performs a UNIX-like **MTIO** operation to the tape device. The type of operation and operation count is set in an **MTIO** structure passed to the **ioctl( )** routine. The **MTIO** operations are defined in Table 5-6.

The **MTIOCTOP** operation is compatible with the UNIX **MTIOCTOP** operation. The argument passed to **ioctl( )** with **MTIOCTOP** is a pointer to an **MTOP** structure that contains the following two fields:

```
typedef struct mtop
    {
    short mt_op;        /* operation */
    int  mt_count;      /* number of operations */
    } MTOP;
```

The **mt_op** field contains the type of **MTIOCTOP** operation to perform. These operations are defined in Table 5-6. The **mt_count** field contains the number of times the operation defined in **mt_op** should be performed.

Table 5-6 **MTIOCTOP Operations**

Function	Value	Meaning
MTWEOF	**0**	Writes an end-of-file record or "file mark."
MTFSF	**1**	Forward-spaces over a file mark.
MTBSF	**2**	Backward-spaces over a file mark.
MTFSR	**3**	Forward-spaces over a data block.
MTBSR	**4**	Backward space over a data block.
MTREW	**5**	Rewinds the tape device to the beginning-of-medium.

Table 5-6 **MTIOCTOP Operations** *(Continued)*

Function	Value	Meaning
MTOFFL	**6**	Rewinds and puts the drive offline.
MTNOP	**7**	No operation; sets status in the **SEQ_DEV** structure only.
MTRETEN	**8**	Re-tensions the tape (cartridge tape only).
MTERASE	**9**	Erases the entire tape.
MTEOM	**10**	Positions the tape to end-of-media.
MTNBSF	**11**	Backward-spaces file to the beginning-of-medium.

5.6 CD-ROM File System: cdromFs

The cdromFs library, **cdromFsLib**, lets applications read any CD-ROM that is formatted in accordance with ISO 9660 file system standards. To configure VxWorks with cdromFs, add the **INCLUDE_CDROMFS** component to the kernel.

After initializing cdromFs and mounting it on a CD-ROM block device, you can access data on that device using the standard POSIX I/O calls: **open( )**, **close( )**, **read( )**, **ioctl( )**, **readdir( )**, and **stat( )**. The **write( )** call always returns an error.

The cdromFs utility supports multiple drives, multiple open files, and concurrent file access. When you specify a pathname, cdromFS accepts both "/" and "\". However, the backslash is not recommended because it might not be supported in future releases.

CdromFs provides access to CD-ROM file systems using any standard **BLK_DEV** structure. The basic initialization sequence is similar to installing a dosFs file system on a SCSI device.

For information on using **cdromFs( )**, see the *VxWorks API Reference* entry for **cdromFsLib**.

Example 5-9 **Creating and Using a CD-ROM Block Device**

The example below describes the steps for creating a block device for the CD-ROM, creating a **cdromFsLib** device, mounting the filesystem, and accessing the media in the device.

Step 1: Configure Your Environment for the CD-ROM Device

Add the component **INCLUDE_CDROMFS** in your project. Add other required components (like SCSI/ATA depending on the type of device). For more information, see *5.2.2 Configuring Your System*, p.197.

If you are using an ATAPI device, make appropriate modifications to the **ataDrv**, **ataResources[ ]** structure array (if needed). This must be configured appropriately for your hardware platform.

Step 2: Create a Block Device

Based on the type of device, use the appropriate create routine and create a block device. Following is an example for an ATAPI master device upon the secondary ATA controller:

```
-> pBlkd = ataDevCreate(1, 0, 0, 0)
new symbol "pBlkd" added to symbol table.
pBlkd = 0x3fff334: value = 67105604 = 0x3fff344 = pBlkd + 0x10
```

Step 3: Create an Instance of a CD-ROM Device in the I/O System

A block device must already have been created. Call **cdromFsDevCreate( )**, which calls **iosDrvInstall( )** internally. This enters the appropriate driver routines in the I/O driver table.

```
-> cdromFsDevCreate("/cd", pBlkd)
value = 67105456 = 0x3fff2b0

-> devs
drv name
0 /null
1 /tyCo/0
1 /tyCo/1
5 ala-petrient:
6 /vio
7 /cd
value = 25 = 0x19

-> cd "/cd"
value = 0 = 0x0
```

The **cd** command changes the current working directory, without performing I/O, and, therefore, can be called before the media is mounted.

Step 4: Mount the Device

Now, run **cdromFsVolConfigShow( )** to indicate whether the device (created above) is mounted. Executing this command shows that the device is not mounted.

```
-> cdromFsVolConfigShow "/cd"

device config structure ptr     0x3fff2b0
device name                     /cd
bytes per blkDevDrv sector      2048
no volume mounted
value = 18 = 0x12
```

Mount the device in order to access it. Because cdromFs is mounted during the first **open( )** operation, a call to **open( )** or any function that uses **open( )** will mount the device. The **ls** command below both mounts the device and lists its contents.

```
-> ls "/cd"
/cd/.
/cd/..
/cd/INDEX.HTML;1
/cd/INSTRUCT.HTML;1
/cd/MPF
/cd/README.TXT;1
value = 0 = 0x0
```

Step 5: Check the Configuration

You can check the CD-ROM configuration using **cdromFsVolConfigShow( )**:

```
-> cdromFsVolConfigShow "/cd"

device config structure ptr     0x3fff2b0
device name                     /cd
bytes per blkDevDrv sector      2048

Primary directory hierarchy:

standard ID                     :CD001
volume descriptor version       :1
system ID                       :LINUX
volume ID                       :MPF_CD
volume size                     :611622912 = 583 MB
number of logical blocks        :298644 = 0x48e94
volume set size                 :1
volume sequence number          :1
logical block size              :2048
path table size (bytes)         :3476
path table entries              :238
volume set ID                   :
```

```
volume publisher ID            :WorldWide Technolgies

volume data preparer ID        :Kelly Corday

volume application ID          :mkisofs v1.04

copyright file name            :mkisofs v1.04
abstract file name             :mkisofs v1.04
bibliographic file name        :mkisofs v1.04
creation date                  :13.11.1998  14:36:49:00
modification date              :13.11.1998  14:36:49:00
expiration date                :00.00.0000  00:00:00:00
effective date                 :13.11.1998  14:36:49:00
value = 0 = 0x0
```

5.7 The Target Server File System: TSFS

The Target Server File System (TSFS) is designed for development and diagnostic purposes. It is a full-featured VxWorks file system, but the files are actually located on the host system.

The TSFS provides all of the I/O features of the network driver for remote file access (**netDrv**), without requiring any target resources (except those required for communication between the target system and the target server on the host). The TSFS uses a WDB driver to transfer requests from the VxWorks I/O system to the target server. The target server reads the request and executes it using the host file system. When you open a file with TSFS, the file being opened is actually on the host. Subsequent **read( )** and **write( )** calls on the file descriptor obtained from the **open( )** call read from and write to the opened host file.

The TSFS VIO driver is oriented toward file I/O rather than toward console operations as is the Tornado 1.0 VIO driver. TSFS provides all the I/O features that **netDrv** provides, without requiring any target resource beyond what is already configured to support communication between target and target server. It is possible to access host files randomly without copying the entire file to the target, to load an object module from a virtual file source, and to supply the filename to routines such as **moduleLoad( )** and **copy( )**.

Each I/O request, including **open( )**, is synchronous; the calling target task is blocked until the operation is complete. This provides flow control not available in the console VIO implementation. In addition, there is no need for WTX protocol requests to be issued to associate the VIO channel with a particular host file; the information is contained in the name of the file.

Consider a **read( )** call. The driver transmits the ID of the file (previously established by an **open( )** call), the address of the buffer to receive the file data, and the desired length of the read to the target server. The target server responds by issuing the equivalent **read( )** call on the host and transfers the data read to the target program. The return value of **read( )** and any **errno** that might arise are also relayed to the target, so that the file appears to be local in every way.

For detailed information, see the *VxWorks API Reference* entry for **wdbTsfsDrv**.

Socket Support

TSFS sockets are operated on in a similar way to other TSFS files, using **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**. To open a TSFS socket, use one of the following forms of filename:

```
"TCP:hostIP:port"
"TCP:hostname:port"
```

The *flags* and *permissions* arguments are ignored. The following examples show how to use these filenames:

```
fd = open("/tgtsvr/TCP:phobos:6164"0,0)     /* open socket and connect  */
                                            /* to server phobos         */

fd = open("/tgtsvr/TCP:150.50.50.50:6164",0,0)  /* open socket and     */
                                                /*connect to server    */
                                                /* 150.50.50.50        */
```

The result of this **open( )** call is to open a TCP socket on the host and connect it to the target server socket at *hostname* or *hostIP* awaiting connections on *port*. The resultant socket is non-blocking. Use **read( )** and **write( )** to read and write to the TSFS socket. Because the socket is non-blocking, the **read( )** call returns immediately with an error and the appropriate **errno** if there is no data available to read from the socket. The **ioctl( )** usage specific to TSFS sockets is discussed in the *VxWorks API Reference* entry for **wdbTsfsDrv**. This socket configuration allows VxWorks to use the socket facility without requiring **sockLib** and the networking modules on the target.

Error Handling

Errors can arise at various points within TSFS and are reported back to the original caller on the target, along with an appropriate error code. The error code returned is the VxWorks **errno** which most closely matches the error experienced on the host. If a WDB error is encountered, a WDB error message is returned rather than a VxWorks **errno**.

TSFS Configuration

To use the TSFS, your VxWorks-based system must be configured with the **INCLUDE_WDB_TSFS** component in the kernel. This creates the **/tgtsvr** file system.

The target server on the host system must also be configured for TSFS. This involves assigning a root directory on your host to TSFS (see the discussion of the target server **-R** option in *Security Considerations*, p.239). For example, on a PC host you could set the TSFS root to **c:\myTarget\logs**.

Having done so, opening the file **/tgtsvr/logFoo** on the target causes **c:\myTarget\logs\logFoo** to be opened on the host by the target server. A new file descriptor representing that file is returned to the caller on the target.

Security Considerations

While TSFS has much in common with **netDrv**, the security considerations are different. With TSFS, the host file operations are done on behalf of the user that launched the target server. The user name given to the target as a boot parameter has no effect. In fact, none of the boot parameters have any effect on the access privileges of TSFS.

In this environment, it is less clear to the user what the privilege restrictions to TSFS actually are, since the user ID and host machine that start the target server may vary from invocation to invocation. By default, any Tornado tool that connects to a target server which is supporting TSFS has access to any file with the same authorizations as the user that started that target server. However, the target server can be locked (with the **-L** option) to restrict access to the TSFS.

The options which have been added to the target server startup routine to control target access to host files using TSFS include:

-L Lock the target server.

This option restricts access to the server to processes running under the same user ID for UNIX, or the same **WIND_UID** for Windows.

-R Set the root of TSFS.

For example, specifying **-R /tftpboot** prepends this string to all TSFS filenames received by the target server, so that **/tgtsvr/etc/passwd** maps to **/tftpboot/etc/passwd**. If **-R** is not specified, TSFS is not activated and no TSFS requests from the target will succeed. Restarting the target server without specifying **-R** disables TSFS.

-RW Make TSFS read-write.

The target server interprets this option to mean that modifying operations (including file create and delete or write) are authorized. If **-RW** is not specified, the default is read only and no file modifications are allowed.

NOTE: For more information about the target server and the TSFS, see the **tgtsvr** entry in the online *Tornado Tools Reference*. For information about specifying target server options from the Tornado IDE, see the *Tornado User's Reference: Target Manager*.

6
Target Tools

6.1 Introduction

The Tornado development system provides a full suite of development tools that resides and executes on the host machine; this approach conserves target memory and resources. However, there are many situations in which it is desirable to have a target-resident shell, a target-resident dynamic object loader, simple target-resident debug facilities, or a target-resident system symbol table. This chapter discusses the target-resident facilities in detail.

Some situations in which the target based tools may be particularly useful are:

- When debugging a deployed system over a simple serial connection.
- When developing and debugging network protocols, where it is useful to see the target's view of a network.
- To create dynamically configurable systems that can load modules from a target disk or over the network.

The target based tools are partially independent so, for instance, the target shell may be used without the target loader, and vice versa. However, for any of the other individual tools to be completely functional, the system symbol table is required.

In some situations, it may also be useful to use both the host-resident development tools and the target-resident tools at the same time. In this case, additional facilities are required so that both environments maintain consistent views of the system. For more information, see *6.4.4 Using the VxWorks System Symbol Table*, p.266.

This chapter briefly describes these target-resident facilities, as well as providing an overview of the most commonly used VxWorks show routines.

For the most part, the target-resident facilities work the same as their Tornado host counterparts. For more information, see the appropriate chapters of the *Tornado User's Guide*.

6.2 Target-Resident Shell

For the most part, the target-resident shell works the same as the host shell (also known as WindSh or the Tornado shell). For detailed information about the host shell, as well as the differences between the host and target shell, see the *Tornado User's Guide: Shell*. Also see the *VxWorks API Reference* entries for **dbgLib**, **shellLib**, and **usrLib**.

6.2.1 Summarizing the Target and Host Shell Differences

The major differences between the target and host shells are:

- The Tornado host shell provides additional commands.
- Both shells include a C interpreter; the host shell also provides a Tcl interpreter. Both shells provide an editing mode.
- You can have multiple host shells active for any given target; only one target shell can be active for a target at any one time.
- The host shell allows virtual I/O; the target shell does not.
- The host shell is always ready to execute provided that the WDB target agent is included in the system. The target shell, as well as its associated target-resident symbol tables and module loader, must be configured into the VxWorks image by including the appropriate components.
- The target shell's input and output are directed at the same window by default, usually a console connected to the board's serial port.[1] For the host shell, these standard I/O streams are not necessarily directed to the same window as the host shell.

[1] Provided that suitable hardware is available, standard input and output can be redirected once the shell is running with **ioGlobalStdSet( )**.

- The host shell can perform many control and information functions entirely on the host, without consuming target resources.
- The host shell uses host resources for most functions so that it remains segregated from the target. This means that the host shell can operate on the target from the outside. The target shell, however, must act on itself, which means that there are limitations to what it can do. For example, to make breakable calls in the target shell, **sp( )** must be used. In addition, conflicts in task priority may occur while using the target shell.

⚠ **WARNING:** Shell commands must be used in conformance with the routine prototype, or they may cause the system to hang (as for example, using **ld( )** without an argument does).

- The target shell correctly interprets the tilde operator in pathnames, whereas the host shell cannot. For example, the following command executed from the target shell by user **panloki** would correctly locate **/home/panloki/foo.o** on the host system:

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

- When the target shell encounters a string literal ("...") in an expression, it allocates space for the string, including the null-byte string terminator, plus some additional overhead.[2] The value of the literal is the address of the string in the newly allocated storage. For example, the following expression allocates 12-plus bytes from the target memory pool, enters the string in that memory (including the null terminator), and assigns the address of the string to **x**:

  ```
  -> x = "hello there"
  ```

  The following expression can be used to return the memory to the target memory pool (see the **memLib** reference entry for information on memory management):

  ```
  -> free (x)
  ```

  Furthermore, even when a string literal is not assigned to a symbol, memory is still permanently allocated for it. For example, the following expression uses memory that is never freed:

  ```
  -> printf ("hello there")
  ```

[2] The amount of memory allocated is rounded up to the minimum allocation unit for the architecture in question, plus the amount for the header for that block of memory.

This is because if strings were only temporarily allocated, and a string literal were passed to a routine being spawned as a task, by the time the task executed and attempted to access the string, the target shell would have already released (and possibly even reused) the temporary storage where the string was held.

After extended development sessions with the target shell, the cumulative memory used for strings may be noticeable. If this becomes a problem, you must reboot your target.

The host shell also allocates memory on the target if the string is to be used there. However, it does not allocate memory on the target for commands that can be performed at the host level (such as **lkup( )**, **ld( )**, and so on).

6.2.2 Configuring VxWorks With the Target Shell

To create the target shell, you must configure VxWorks to include the **INCLUDE_SHELL** component.

You must also configure VxWorks with components for symbol table support (see *6.4.1 Configuring VxWorks with Symbol Tables*, p.262). Additional shell components include facilities for the following:

INCLUDE_SHELL_BANNER
Display the shell banner.

INCLUDE_DEBUG
Include shell debug facilities.

INCLUDE_DISK_UTIL
Include file utilities, such as **ls** and **cd**.

INCLUDE_SYM_TBL_SHOW
Include *Symbol Table Show Routines*, such as **lkup**.

You may also find it useful to include components for the module loader and unloader (see *6.3.1 Configuring VxWorks with the Loader*, p.251).These components are required for the **usrLib** commands that load and unload modules (see *6.2.4 Loading and Unloading Object Modules from the Target Shell*, p.245).

The shell task (**tShell**) is created with the **VX_UNBREAKABLE** option; therefore, breakpoints cannot be set in this task, because a breakpoint in the shell would make it impossible for the user to interact with the system. Any routine or task that is invoked from the target shell, rather than spawned, runs in the **tShell** context.

Only one target shell can run on a VxWorks system at a time; the target shell parser is not reentrant, because it is implemented using the UNIX tool **yacc**.

6.2.3 Using Target Shell Help and Control Characters

You can type the following command to display help on the shell:

```
-> help
```

Use **dbgHelp** for commands related to debugging.

The following target shell command lists all the available help routines:

```
-> lkup "Help"
```

The target shell has its own set of terminal-control characters, unlike the host shell, which inherits its setting from the host window from which it was invoked. Table 6-1 lists the target shell's terminal-control characters. The first four of these are defaults that can be mapped to different keys using routines in **tyLib** (see also *Tty Special Characters*, p.133).

Table 6-1 **Target Shell Terminal Control Characters**

Command	Description
CTRL+C	Aborts and restarts the shell.
CTRL+H	Deletes a character (backspace).
CTRL+Q	Resumes output.
CTRL+S	Temporarily suspends output.
CTRL+U	Deletes an entire line.
CTRL+X	Reboots (trap to the ROM monitor).
ESC	Toggles between input mode and edit mode (**vi** mode only).

The shell line-editing commands are the same as they are for the host shell.

6.2.4 Loading and Unloading Object Modules from the Target Shell

Object modules can be dynamically loaded into a running VxWorks system with the module loader. The following is a typical load command from the shell, in which the user downloads **appl.o** to the **appBucket** domain:

```
[appBucket] -> ld < /home/panloki/appl.o
```

The **ld( )** command loads an object module from a file, or from standard input, into a specified protection domain. External references in the module are resolved during loading.

Once an application module is loaded into target memory, subroutines in the module can be invoked directly from the shell, spawned as tasks, connected to an interrupt, and so on. What can be done with a routine depends on the flags used to download the object module (visibility of global symbols or visibility of all symbols). For more information about **ld**, see the *VxWorks API Reference* entry for **usrLib**.

Modules can be reloaded with **reld( )**, which unloads the previously loaded module of the same name before loading the new version. For more information about **reld**, see the *VxWorks API Reference* entry for **unldLib**.

Undefined symbols can be avoided by loading the modules in the appropriate order. Linking independent files before download can be used to avoid unresolved references if there are circular references between them, or if the number of modules is unwieldy. The static linker **ld***arch* can be used to link interdependent files, so that they can only be loaded and unloaded as a unit.

Unloading a code module removes its symbols from the target's symbol table, removes the code module descriptor and section descriptor(s) from the module list, and removes its section(s) from the memory partition(s) that held them.

For information about features of the target loader and unloader, see *6.3 Target-Resident Loader*, p.250.

6.2.5 Debugging with the Target Shell

The target shell includes the same task level debugging utilities as the host shell if VxWorks has been configured with the **INCLUDE_DEBUG** component. For details on the debugging commands available, see the *Tornado User's Guide: Shell* and the *VxWorks API Reference* entry for **dbgLib**.

You are not permitted to use system mode debug utilities with the target shell.

6.2.6 Aborting Routines Executing from the Target Shell

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine can loop excessively, suspend, or wait on a semaphore. This can happen because of errors in the arguments specified in the invocation, errors in the implementation of the routine, or oversight regarding the consequences of calling the routine. In such cases it is usually possible to abort and restart the target shell task. This is done by pressing the special target-shell abort character on the keyboard, **CTRL+C** by default. This causes the target shell task to restart execution at its original entry point. Note that the abort key can be changed to a character other than **CTRL+C** by calling **tyAbortSet( )**.

When restarted, the target shell automatically reassigns the system standard input and output streams to the original assignments they had when the target shell was first spawned. Thus any target shell redirections are canceled, and any executing shell scripts are aborted.

The abort facility works only if the following are true:

- **dbgInit( )** has been called (see *6.2.5 Debugging with the Target Shell*, p.246).
- **excTask( )** is running (see the *Tornado User's Guide: Configuration and Build*).
- The driver for the particular keyboard device supports it (all VxWorks-supplied drivers do).
- The device's abort option is enabled. This is done with an **ioctl( )** call, usually in the root task in **usrConfig.c**. For information on enabling the target shell abort character, see *tty Options*, p.132.

Also, you may occasionally enter an expression that causes the target shell to incur a fatal error such as a bus/address error or a privilege violation. Such errors normally result in the suspension of the offending task, which allows further debugging.

However, when such an error is incurred by the target shell task, VxWorks automatically restarts the target shell, because further debugging is impossible without it. Note that for this reason, as well as to allow the use of breakpoints and single-stepping, it is often useful when debugging to spawn a routine as a task instead of just calling it directly from the target shell.

When the target shell is aborted for any reason, either because of a fatal error or because it is aborted from the terminal, a task trace is displayed automatically. This trace shows where the target shell was executing when it died.

Note that an offending routine can leave portions of the system in a state that may not be cleared when the target shell is aborted. For instance, the target shell might have taken a semaphore, which cannot be given automatically as part of the abort.

6.2.7 Using a Remote Login to the Target Shell

Remote Login From Host: telnet *and* rlogin

When VxWorks is first booted, the target shell's terminal is normally the system console. You can use **telnet** to access the target shell from a host over the network if you select **INCLUDE_TELNET** for inclusion in the project facility VxWorks view (see *Tornado User's Guide: Projects*). Defining **INCLUDE_TELNET** creates the **tTelnetd** task. To access the target shell over the network, enter the following command from the host (*targetname* is the name of the target VxWorks system):

```
% telnet "targetname"
```

UNIX host systems also use **rlogin** to provide access to the target shell from the host. Select **INCLUDE_RLOGIN** for inclusion in the project facility VxWorks view to create the **tRlogind** task. However, note that VxWorks does not support **telnet** or **rlogin** access from the VxWorks system to the host.

A message is printed on the system console indicating that the target shell is being accessed via **telnet** or **rlogin**, and that it is no longer available from its console.

If the target shell is being accessed remotely, typing at the system console has no effect. The target shell is a single-user system—it allows access either from the system console or from a single remote login session, but not both simultaneously. To prevent someone from remotely logging in while you are at the console, use the routine **shellLock( )** as follows:

```
-> shellLock 1
```

To make the target shell available again to remote login, enter the following:

```
-> shellLock 0
```

To end a remote-login target shell session, call **logout( )** from the target shell. To end an **rlogin** session, type **TILDE** and **DOT** as the only characters on a line:

```
-> ~.
```

Remote Login Security

You can be prompted to enter a login user name and password when accessing VxWorks remotely:

```
VxWorks login: user_name
Password: password
```

The remote-login security feature is enabled by selecting **INCLUDE_SECURITY** for inclusion in the project facility VxWorks view. The default login user name and password provided with the supplied system image is *target* and *password*. You can change the user name and password with the **loginUserAdd( )** routine, as follows:

```
-> loginUserAdd "fred", "encrypted_password"
```

To obtain *encrypted_password*, use the tool **vxencrypt** on the host system. This tool prompts you to enter your password, and then displays the encrypted version.

To define a group of login names, include a list of **loginUserAdd( )** commands in a startup script and run the script after the system has been booted. Or include the list of **loginUserAdd( )** commands to the file **usrConfig.c**, then rebuild VxWorks.

NOTE: The values for the user name and password apply only to remote login into the VxWorks system. They do not affect network access from VxWorks to a remote system; See *VxWorks Network Programmer's Guide: rlogin and telnet, Host Access Applications*.

The remote-login security feature can be disabled at boot time by specifying the flag bit 0x20 (**SYSFLAG_NO_SECURITY**) in the *flags* parameter on the boot line (see *Tornado Getting Started*). This feature can also be disabled by deselecting **INCLUDE_SECURITY** in the project facility VxWorks view.

6.2.8 Distributing the Demangler

The Wind River Target Shell provided as part of the suite of VxWorks development tools includes an unlinked copy of the demangler library. This library is licensed under the GNU Library General Public License Version 2, June 1991 - a copy of which can be found at **www.gnu.org** or by downloading sources to your GNU toolchain from the WindSurf site. Under this license the demangler library can be used without restriction during development, but VxWorks developers should be aware of the restrictions provided within the Library General Public License should the library be linked with application code as a derivative work and distributed to third parties.

If VxWorks developers wish to distribute the Wind River Target Shell while avoiding the Library General Public License restrictions, then the demangler library should be excluded from the project.

To remove the demangler from a kernel built with the Tornado project facility simply remove the "C++ symbol demangler" component.

To remove the demangler from a kernel built from a BSP (command line build) define the macro **INCLUDE_NO_CPLUS_DEMANGLER** in **config.h**.

Excluding this library will not affect the operation of the Wind River Target Shell but will reduce the human readability of C++ identifiers and symbols.

6.3 Target-Resident Loader

VxWorks lets you add code to a target system at run-time. This operation, called *loading*, or *downloading*, allows you to install target applications or to extend the operating system itself.

The downloaded code can be a set of functions, meant to be used by some other code (the equivalent of a library in other operating systems), or it can be an application, meant to be executed by a task or a set of tasks. The units of code that can be downloaded are referred to as object modules.

The ability to load individual object modules brings significant flexibility to the development process, in several different ways. The primary use of this facility during development is to unload, recompile, and reload object modules under development. The alternative is to link the developed code into the VxWorks image, to rebuild this image, and to reboot the target, every time the development code must be recompiled.

The loader also enables you to dynamically extend the operating system, since once code is loaded, there is no distinction between that code and the code that was compiled into the image that booted.

Finally, you can configure the loader to optionally handle memory allocation, on a per-load basis, for modules that are downloaded. This allows flexible use of the target's memory. The loader can either dynamically allocate memory for downloaded code, and free that memory when the module is unloaded; or, the caller can specify the addresses of memory that has already been allocated. This allows the user more control over the layout of code in memory. For more information, see *6.3.5 Specifying Memory Locations for Loading Objects*, p.255.

The VxWorks target loader functionality is provided by two components: the loader proper, which installs the contents of object modules in the target system's memory; and the unloader, which uninstalls object modules. In addition, the loader relies on information provided by the system symbol table.

NOTE: The target-resident loader is often confused with the bootloader, which is used to install the kernel image in memory. Although these two tools perform similar functions, and share some support code, they are separate entities. The bootloader loads only complete images, and does not perform relocations.

6.3.1 Configuring VxWorks with the Loader

By default, the loader is not included in the VxWorks image. To use the target loader, you must configure VxWorks with the **INCLUDE_LOADER** component. The **INCLUDE_LOADER** component is discussed further in subsequent sections and in the **loadLib** entry in *the VxWorks API Reference*. Adding this component automatically includes several other components that, together, provide complete loader functionality:

INCLUDE_MODULE_MANAGER
Provides facilities for managing loaded modules and obtaining information about them. For more information, see the entry for **moduleLib** in the *VxWorks Reference Manual*.

INCLUDE_SYM_TBL
Provides facilities for storing and retrieving symbols. For more information, see *6.4 Target-Resident Symbol Tables*, p.261 and the entry for **symLib** in the *VxWorks Reference Manual*. For information about configuration of the symbol table, see the *6.4.1 Configuring VxWorks with Symbol Tables*, p.262.

INCLUDE_SYM_TBL_INIT
Specifies a method for initializing the system symbol table.

Including the loader does not automatically include the unloader. To add the unloader to your system, include the following component:

INCLUDE_UNLOADER
Provides facilities for unloading modules. For additional information on unloading, see **unldLib** entry in the *VxWorks API Reference*.

INCLUDE_SYM_TBL_SYNC
Provides host-target symbol table and module synchronization.

⚠ **CAUTION:** If you want to use the target-resident symbol tables and loader in addition to the Tornado host tools, you must configure VxWorks with the **INCLUDE_SYM_TBL_SYNC** component to provide host-target symbol table and module synchronization. For more information, see *6.4.4 Using the VxWorks System Symbol Table*, p.266.

6.3.2 Target-Loader API

Table 6-2 and Table 6-3 list the API routines and shell commands available for module loading and unloading.

Table 6-2 **Routines for Loading and Unloading Object Modules**

Routine	Description
loadModule( )	Loads an object module.
loadModuleAt( )	Loads an object module into a specific memory location.
unldByModuleId( )	Unloads an object module by specifying a module ID.
unldByNameAndPath( )	Unloads an object module by specifying a name and path.

Note that all of the loader routines can be called directly from the shell or from code. The *shell commands*, however, are intended to be used only from the shell, not from within programs.[3] In general, shell commands handle auxiliary operations, such as opening and closing a file; they also print their results and any error messages to the console.

Table 6-3 **Shell Command for Loading and Unloading Object Modules**

Command	Description
ld( )	Loads an object module.
reld( )	Unloads and reloads an object module by specifying a filename or module ID.
unld( )	Unloads an object module by specifying a filename or module ID.

The use of some of these routines and commands is discussed in the following sections.

[3] In future releases, calling shell commands programmatically may not be supported.

For detailed information, see the **loadLib**, **unldLib**, and **usrLib** entries in the *VxWorks API Reference*, as well as *6.3.3 Summary List of Loader Options*, p.253.

6.3.3 Summary List of Loader Options

The loader's behavior can be controlled using load flags passed to **loadLib** and **unldLib** routines. These flags can be combined (using an **OR** operation), although some are mutually exclusive. Table 6-4, Table 6-5, and Table 6-5 group these options by category.

Table 6-4 **Loader Options for Symbol Registration**

Option	Hex Value	Description
LOAD_NO_SYMBOLS	0x2	No symbols from the module are registered in the system's symbol table. Consequently, linkage against the code module is not possible. This option is useful for deployed systems, when the module is not supposed to be used in subsequent link operations.
LOAD_LOCAL_SYMBOLS	0x4	Only local (private) symbols from the module are registered in the system's symbol table. No linkage is possible against this code module92s public symbols. This option is not very useful by it self, but is one of the base options for **LOAD_ALL_SYMBOLS**.
LOAD_GLOBAL_SYMBOLS	0x8	Only global (public) symbols from the module are registered in the system's symbol table. No linkage is possible against this code module's private symbols. This is the loader's default when the **loadFlags** parameter is left as **NULL**.
LOAD_ALL_SYMBOLS	0xC	Local and global symbols from the module are registered in the system's symbol table. This option is useful for debugging.

Table 6-5 **Loader Option for Code Module Visibility**

Option	Hex Value	Description
HIDDEN_MODULE	0x10	The code module is not visible from the **moduleShow( )** routine or the Tornado tools. This is useful on deployed systems when an automatically loaded module should not be detectable by the user.

Table 6-6 **Loader Options for Resolving Common Symbols**

Option	Hex Value	Description
LOAD_COMMON_MATCH_NONE	0x100	Keeps common symbols isolated, visible from the object module only. This option prevents any matching with already-existing symbols. Common symbols are added to the symbol table unless **LOAD_NO_SYMBOLS** is set. This is the default option.
LOAD_COMMON_MATCH_USER	0x200	Seeks a matching symbol in the system symbol table, but considers only symbols in user modules, not symbols that were in the original booted image. If no matching symbol exists, this option behaves like **LOAD_COMMON_MATCH_NONE**.
LOAD_COMMON_MATCH_ALL	0x400	Seeks a matching symbol in the system symbol table, considering all symbols. If no matching symbol exists, this option behaves like **LOAD_COMMON_MATCH_NONE**.

If several matching symbols exist for options **LOAD_COMMON_MATCH_USER** and **LOAD_COMMON_MATCH_ALL**, the order of precedence is: symbols in the **bss** segment, then symbols in the **data** segment. If several matching symbols exist within a single segment type, the symbol most recently added to the target server symbol table is used.

Table 6-7 **Unloader Option for Breakpoints and Hooks**

Option	Hex Value	Description
UNLD_KEEP_BREAKPOINTS	0x1	T he breakpoints are left in place when the code module is unloaded. This is useful for debugging, as all breakpoints are otherwise removed from the system when a module is unloaded.

6.3.4 Loading C++ Modules

The files produced by the toolchains are not sufficiently processed for the loader to handle them properly. For instructions on preparing a C++ file for download, see section *7.3.1 Munching C++ Application Modules*, p.278.

C++ modules may include code that must be executed automatically when they are loaded and unloaded. This code consists of constructors and destructors for static instances of classes.

Constructors must be executed before any other code in the code module is executed. Similarly, destructors must be executed after the code module is no longer to be used, but before the module is unloaded. In general, executing constructors at load time and destructors at unload time is the simplest way to handle them. However, for debugging purposes, a user may want to decouple the execution of constructors and destructors from the load and unload steps.

For this reason, the behavior of the loader and unloader regarding constructors and destructors is configurable. When the C++ strategy is set to **AUTOMATIC** (1), the constructors and destructors are executed at load and unload time. When the C++ strategy is set to **MANUAL** (0), constructors and destructors are not executed by the loader or unloader. When the C++ strategy is **MANUAL**, the functions **cplusCtors( )** and **cplusDtors( )** can be used to execute the constructors or destructors for a particular code module.

The function **cplusXtorSet( )** is available to change the C++ strategy at runtime. The default setting is **AUTOMATIC**.

For more information, see the **cplusLib** and **loadLib** entries in the *VxWorks API Reference*.

6.3.5 Specifying Memory Locations for Loading Objects

By default, the loader allocates the memory necessary to hold a code module. It is also possible to specify where in memory any or all of the **text**, **data**, and **bss** segments of an object module should be installed using the **loadModuleAt( )** command. If an address is specified for a segment, then the caller must allocate sufficient space for the segment at that address before calling the load routine. If no addresses are specified, the loader allocates one contiguous area of memory for all three of the segments.

For any segment that does not have an address specified, the loader allocates the memory (using **memPartAlloc( )** or, for aligned memory, using **memalign( )**). The base address can also be set to the value **LOAD_NO_ADDRESS**, in which case the loader replaces the **LOAD_NO_ADDRESS** value with the actual base address of the segment once the segment is installed in memory.

When working with architectures that use the ELF object module format, an additional complication arises. The basic unit of information in a relocatable ELF object file is a section. In order to minimize memory fragmentation, the loader gathers sections so that they form the logical equivalent of an ELF segment. For simplicity, these groups of sections are also referred to as segments. For more information, see *ELF object module format*, p.257).

The VxWorks loader creates three segments: **text**, **data**, and **bss**. When gathering sections together to form segments, the sections are placed into the segments in the same order in which they occur in the ELF file. It is sometimes necessary to add extra space between sections to satisfy the alignment requirements of all of the sections. When allocating space for one or more segments, care must be taken to ensure that there is enough space to permit all of the sections to be aligned properly. (The alignment requirement of a section is given as part of the section description in the ELF format. The binary utilities **readelf***arch* and **objdump***arch* can be used to obtain the alignment information.)

In addition, the amount of padding required between sections depends on the alignment of the base address. To ensure that there will be enough space without knowing the base address in advance, allocate the block of memory so that it is aligned to the maximum alignment requirement of any section in the segment. So, for instance, if the data segment contains sections requiring 128 and 264 byte alignment, in that order, allocate memory aligned on 264 bytes.

The other object module formats used in VxWorks (**a.out** and PECOFF) have basic units of information that correspond more closely to the VxWorks model of a single unit each of text, data and bss. Therefore when working with architectures that use either the **a.out** or PECOFF file formats, this problem of allocating extra space between sections does not arise.

The unloader can remove the sections wherever they have been installed, so no special instructions are required to unload modules that were initially loaded at specific addresses. However, if the base address was specified in the call to the loader, then, as part of the unload, unloader does not free the memory area used to hold the segment. This allocation was performed by the caller, and the de-allocation must be as well.

6.3.6 Constraints Affecting Loader Behavior

The following sections describe the criteria used to load modules.

Relocatable Object Files

A relocatable file is an object file for which **text** and **data** sections are in a transitory form, meaning that some addresses are not yet known. An executable file is one which is fully linked and ready to run at a specified address. In many operating systems, relocatable object modules are an intermediate step between source (**.c**, **.s**, **.cpp**) files and executable files, and only executable files may be loaded and run.

The relocatable files produced by the toolchains are labeled with a **.o** extension.

However, in VxWorks, relocatable (**.o**) files are used for application code, for the following reason. To construct an executable image for download, the program's execution address and the addresses of externally defined symbols, such as library routines, must be known. Since the layout of the VxWorks image and downloaded code in memory are so flexible, these items of information are not available to a compiler running on a host machine. Therefore, the code handled by the target-resident loader must be in relocatable form, rather than an executable.

Once installed in the system's memory, the entity composed of the object module's code, data, and symbols is called a code module. For information about installed code modules, see the individual routine entries under **moduleLib** in the *VxWorks API Reference*.

Object Module Formats

There are several standard formats for both relocatable and executable object files. In the Tornado development environment, there is a single preferred object module format for each supported target architecture. For most architectures, the preferred format is now Executable and Linkable Format (ELF). Exceptions are the 68K toolchain, which produces **a.out** object files, and the NT simulator toolchain, which produces **pecoff** object files.

The VxWorks loader can handle most object files generated by the supported toolchains.

NOTE: Not all possible relocation operations that are supported by an architecture are necessarily supported by the loader. This will usually only affect users writing assembly code.

ELF object module format

An relocatable ELF object file is essentially composed of two categories of elements: the headers and the sections. The headers describe the sections, and the sections contain the actual **text** and **data** to be installed.

An executable ELF file is a collection of segments, which are aggregations of sections. The VxWorks loader performs an aggregation step on the relocatable object files that is similar to the process carried out by toolchains when producing an executable ELF file. The resulting image consists of one **text** segment, one **data** segment, and one **bss** segment. (A general ELF executable file may have more than one segment of each type, but the VxWorks loader uses the simpler model of at most one segment of each type.) The loader installs the following categories of sections in the system's memory:

- **text** sections that hold the application's instructions
- **data** sections that hold the application's initialized data
- **bss** sections that hold the application's un-initialized data
- read-only data sections that hold the application's constant data

Read-only data sections are placed in the text segment by the loader.

a.out Object Module Format

Object files in the **a.out** file format are fairly simple, and are already very close to the VxWorks abstraction of code modules. Similarly to ELF files, they contain both headers and sections. In other literature on file formats, the 'sections' of an **a.out** file are sometimes referred to as sections and sometimes as segments, even within the same work, depending on the context. Since only one **text** section, one **data** section, and one **bss** section are permitted in an **a.out** file, **a.out** sections are essentially equivalent segments, for our purposes.

The **text**, **data**, and **bss** sections of the **a.out** object module become the **text**, **data** and **bss** segments of the loaded code module.

PECOFF Object Module Format

The PECOFF object module format is only used in VxWorks for the Windows simulator architecture.

The basic unit of information in a PECOFF object module is also called a section. The allowed size a section is limited by the PECOFF file format. Therefore, it is possible, although not common, to have more than one **text** or **data** section in a PECOFF file.

The VxWorks target loader only handles PECOFF object files that have at most one **text** section, one **data** section, and one **bss** section. These sections become the **text**, **data**, and **bss** segments of VxWorks code module.

Linking and Reference Resolution

The VxWorks loader performs some of the same tasks as a traditional linker in that it prepares the code and data of an object module for the execution environment. This includes the linkage of the module's code and data to other code and data.

The loader is unlike a traditional linker in that it does this work directly in the target system's memory, and not in producing an output file.

In addition, the loader uses routines and variables that already exist in the VxWorks system, rather than library files, to relocate the object module that it loads. The system symbol table (see *6.4.4 Using the VxWorks System Symbol Table*, p.266) is used to store the names and addresses of functions and variables already installed in the system.This has the side effect that once symbols are installed in the system symbol table, they are available for future linking by any module that is loaded. Moreover, when attempting to resolve undefined symbols in a module, the loader uses all global symbols compiled into the target image, as well as all global symbols of previously loaded modules. As part of the normal load process, all of the global symbols provided by a module are registered in the system symbol table. You can override this behavior by using the **LOAD_NO_SYMBOLS** load flag (see Table 6-4).

The system symbol table allows name clashes to occur. For example, suppose a symbol named **func** exists in the system. A second symbol named **func** is added to the system symbol table as part of a load. From this point on, all links to **func** are to the most recently loaded symbol. See also, *6.4.1 Configuring VxWorks with Symbol Tables*, p.262.

The Sequential Nature of Loading

The VxWorks loader loads code modules in a sequential manner. That is, a separate load is required for each separate code module. Suppose a user has two code modules named **A_module** and **B_module**, and **A_module** references symbols that are contained in **B_module**. The user may either use the host-resident linker to combine **A_module** and **B_module** into a single module, or may load **B_module** first, and then load **A_module**.

When code modules are loaded, they are irreversibly linked to the existing environment; meaning that, once a link from a module to an external symbol is created, that link cannot be changed without unloading and reloading the module.

Therefore dependencies between modules must be taken into account when modules are loaded to ensure that references can be resolved for each new module, using either code compiled into the VxWorks image or modules that have already been loaded into the system.

Failure to do so results in incompletely resolved code, which retains references to undefined symbols at the end of the load process. For diagnostic purposes, the loader prints a list of missing symbols to the console. This code should not be executed, since the behavior when attempting to execute an improperly relocated instruction is not predictable.

Normally, if a load fails, the partially installed code is removed. However, if the only failure is that some symbols are unresolved, the code is not automatically unloaded. This allows the user to examine the result of the failed load, and even to execute portions of the code that are known to be completely resolved. Therefore, code modules that have unresolved symbols must be removed by a separate unload command, **unld( )**.

Note that the sequential nature of the VxWorks loader means that unloading a code module which has been used to resolve another code module may leave references to code or data which are no longer available. Execution of code holding such dangling references may have unexpected results.

Resolving Common Symbols

Common symbols provide a challenge for the VxWorks loader that is not confronted by a traditional linker. Consider the following example:

```
#include <stdio.h>

int willBeCommon;

void main (void) {}
    {
    ...
    }
```

The symbol **willBeCommon** is uninitialized, so it is technically an undefined symbol. Many compilers will generate a 'common' symbol in this case.

ANSI C allows multiple object modules to define uninitialized global symbols of the same name. The linker is expected to consistently resolve the various modules references to these symbols by linking them against a unique instance of the symbol. If the different references specify different sizes, the linker should define a single symbol with the size of the largest one and link all references against it.

This is not a difficult task when all of the modules are present at the time linkage takes place. However, because VxWorks modules are loaded sequentially, in independent load operations, the loader cannot know what modules will be loaded or what sets of common symbols might be encountered in the future.

Therefore, the VxWorks target loader has an option that allows control over how common symbols are linked. The default behavior for the **loadLib** API functions is to treat common symbols as if there were no previous matching reference

(**LOAD_COMMON_MATCH_NONE**). The result is that every loaded module has its own copy of the symbol. The other possible behaviors are to permit common symbols to be identified either with any matching symbol in the symbol table (**LOAD_COMMON_MATCH_ALL**), or with any matching symbol that was not in the original booted image (**LOAD_COMMON_MATCH_USER**).

The option to specify matching of common symbols may be set in each call using the **loadLib** API. Extreme care should be used when mixing the different possible common matching behaviors for the loader. It is much safer to pick a single matching behavior and to use it for all loads. For detailed descriptions of the matching behavior under each option, see Table 6-6.

NOTE: Note that the shell load command, **ld**, has a different mechanism for controlling how common symbols are handled and different default behavior. For details, see the reference entry for **usrLib**.

6.4 Target-Resident Symbol Tables

A symbol table is a data structure that stores information that describes the routines, variables, and constants in all modules, and any variables created from the shell. There is a symbol table library, which can be used to manipulate the two different types of symbol tables: a user system table and a system symbol table, which is the most commonly used.

Symbol Entries

Each symbol in the table comprises three items:

name
The name is a character string derived from the name in the source code.

value
The value is usually the address of the element that the symbol refers to: either the address of a routine, or the address of a variable (that is, the address of the contents of the variable).

type

The type is provides additional information about the symbol. For symbols in the system symbol table, it is one of the types defined in *installDir***/target/h/symbol.h**; for example, **SYM_UNDF**, **SYM_TEXT**, and so on. For user symbol tables, this field can be user-defined.

Symbol Updates

The symbol table is updated whenever modules are loaded into, or unloaded from, the target. You can control the precise information stored in the symbol table by using the loader options listed in Table 6-4.

Searching the Symbol Library

You can easily search all symbol tables for specific symbols. To search from the shell, use **lkup( )**. For details, see the **lkup( )** reference entry. To search programmatically, use the symbol library API's, which can be used to search the symbol table by address, by name, and by type, and a function that may be used to apply a user-supplied function to every symbol in the symbol table. For details, see the **symLib** reference entry.

6.4.1 Configuring VxWorks with Symbol Tables

The basic configuration is required for all symbol tables. This configuration provides the symbol table library, and is sufficient for a user symbol table. However, it is *not sufficient* for creating a system symbol table. This section describes both the basic configuration, and the additional configuration necessary to create a system symbol table.

For information about user symbol tables, see *6.4.6 Creating User Symbol Tables*, p.268. For information about the system symbol table, see *6.4.4 Using the VxWorks System Symbol Table*, p.266.

Basic Configuration

The most basic configuration for a symbol table is to include the component, **INCLUDE_SYM_TBL**. This provides the basic symbol table library, **symLib**, (which is not equivalent to the system symbol table) and sufficient configuration for creating user symbol tables. The symbol table library component includes configuration options, which allow you to modify the symbol table width and control whether name clashes are permitted.

- **Hash Table Width**

  The **INCLUDE_SYM_TBL** component includes a configuration parameter that allows the user to change the default symbol table "width". The parameter **SYM_TBL_HASH_SIZE_LOG2** defines the "width" of the symbol table's hash table. It takes a positive value that is interpreted as a power of two. The default value for **SYM_TBL_HASH_SIZE_LOG2** is 8; thus, the default "width" of the symbol table is 256. Using smaller values requires less memory, but degrades lookup performance, so the search takes longer on average. For information about changing this parameter, see the *Tornado User's Guide: Configuration and Build*.

- **Name Clash Handling**

  The system symbol table, **sysSymTbl**, can be configured to allow or disallow name clashes. The flags used to call **symTblCreate( )** determine whether or not duplicate names are permitted in a symbol table. If set to **FALSE**, only one occurrence of a given symbol name is permitted. When name clashes are permitted, the most recently added symbol of several with the same name is the one that is returned when searching the symbol table by name.

System Symbol Table Configuration

To include information about the symbols present in the kernel—and therefore to enable the shell, loader, and debugging facilities to function properly—a system symbol table must be created and initialized. You create a system symbol table at initialization time by including one of the following components:

- **INCLUDE_STANDALONE_SYM_TBL .** Creates a built-in system symbol table, in which both the system symbol table and the VxWorks image are contained in the same module. This type of symbol table is described in *6.4.2 Creating a Built-In System Symbol Table*, p.264.

- **INCLUDE_NET_SYM_TBL .** Creates an separate system symbol table as a **.sym** file that is downloaded to the VxWorks system. This type of symbol table is described in *6.4.3 Creating a Loadable System Symbol Table*, p.265.

When the system symbol table is first created at system initialization time, it contains no symbols. Symbols must be added to the table at run-time. Each of these components handles the process of adding symbols differently.

NOTE: When building in a BSP directory, rather than using the project facility, including the component **INCLUDE_STANDALONE_SYM_TBL** is not sufficient. Instead, build the make target "vxWorks.st." This configuration is for a "standalone" image, and does not initialize the network facilities on the target. To use both the standalone symbol table and the target network facilities, configure your image using the project facility. For details, see the *Tornado User's Guide: Projects*.

6.4.2 Creating a Built-In System Symbol Table

A built-in system symbol table copies information into wrapper code, which is then compiled and linked into the kernel when the system is built.

Generating the Symbol Information

A built-in system symbol table relies on the **makeSymTbl** utility to obtain the symbol information. This utility uses the gnu utility **nm***arch* to generate information about the symbols contained in the image. Then it processes this information into the file **symTbl.c** that contains an array, **standTbl**, of type **SYMBOL** described in *Symbol Entries*, p. 261. Each entry in the array has the symbol **name** and **type** fields set. The address (**value**) field is not filled in by **makeSymTbl**.

Compiling and Linking the Symbol File

The **symTbl.c** file is treated as a normal **.c** file, and is compiled and linked with the rest of the VxWorks image. As part of the normal linking process, the toolchain linker fills in the correct address for each global symbol in the array. When the build completes, the symbol information is available in the image as a global array of VxWorks **SYMBOL**'s. After the kernel image is loaded into target memory at system initialization, the information from the global **SYMBOL** array is used to construct the system symbol table.

The definition of the **standTbl** array can be found in the following files (only after a build). These files are generated as part of the build of the VxWorks image:

installDir/**target/config/***BSPname***/symTbl.c**
for images build directly from a BSP directory

installDir/**target/proj/***projDir*/*buildDir***/symTbl.c**
for images using the project facility

Advantages of Using a Built-in System Symbol Table

Although this method creates a resulting VxWorks image (module file) that is larger than it would be without symbols, the built-in symbol table has advantages, some of which are useful when the download link is slow. These advantages are:

- It saves memory if you are not otherwise using the target loader, because it does not require the target loader (and associated components) to be configured into the system.
- It does not require the target to have access to a host (unlike the downloadable system symbol table).
- It is faster than loading two files (the image and **.sym** files) because network operations[4] on a file take longer than the data transfer to memory.
- It is useful in deployed ROM-based systems that have no network connectivity, but require the shell as user interface.

6.4.3 Creating a Loadable System Symbol Table

A loadable symbol table is built into a separate object module file (**vxWorks.sym** file). This file is downloaded to the system separately from the system image, at which time the information is copied into the symbol table.

Creating the .sym *File*

The loadable system symbol table uses a **vxWorks.sym** file, rather than the **symTbl.c** file. The **vxWorks.sym** file is created by using the **objcopy** utility to strip all sections, except the symbol information, from the final VxWorks image. The resulting symbol information is placed into a file named **vxWorks.sym**.

For architectures using the ELF object module format for the VxWorks image, the **vxWorks.sym** file is also in the ELF format, and contains only a **SYMTAB** section

[4] That use **open( )**, **seek( )**, **read( )**, and **close( )**.

and a **STRTAB** section. In general, the file **vxWorks.sym** is in the same object format as other object files compiled for the target by the installed toolchain.

Loading the .sym *File*

During boot and initialization, the **vxWorks.sym** file is downloaded using the loader, which directly calls **loadModuleAt( )**. To download the **vxWorks.sym** file, the loader uses the current default device, which is described in *4.2.1 Filenames and the Default Device*, p.109.

To download the VxWorks image, the loader also uses the default device, as is current at the time of that download. Therefore, the default device used to download the **vxWorks.sym** file may, or may not, be the same device. This is because the default device can be set, or reset, by other initialization code that runs. This modification can happen after the VxWorks image is downloaded, but before the symbol table is downloaded.

Nevertheless, in standard VxWorks configurations, that do not include customized system initialization code, the default device at the time of the download of the **vxWorks.sym**, is usually set to one of the network devices, and using either **rsh** or **ftp** as the protocol.

Advantages of Using the Loadable System Symbol Table

Using a downloadable symbol table requires that the loader component be present. Therefore, the system must have sufficient memory for the loader and its associated components. A summary of the advantages of using the loadable system symbol table are:

- It is convenient, if the loader is required for other purposes as well.
- It is slightly faster to build than a built-in system symbol table.

6.4.4 Using the VxWorks System Symbol Table

Once it is initialized, the VxWorks system symbol table includes a complete list of the names and addresses of all global symbols in the compiled image that is booted. This information is needed on the target to enable the full functionality of the 'target tools' libraries. For example:

The target tools maintain the system symbol table so that it contains up to date name and address information for all of the code statically compiled into the system or dynamically downloaded. (You can use the **LOAD_NO_SYMBOLS** option to *hide* loaded modules, so that their symbols do not appear in the system symbol table; see Table 6-5).

If the facilities provided by the symbol table library are needed for non-operating system code, another symbol table may be created and manipulated using the symbol library.

Symbols are dynamically added to, and removed from, the system symbol table each time any of the following occurs:

- as modules are loaded and unloaded
- as variables are dynamically created using the shell
- by the wdb agent when synchronizing symbol information with the host

The exact dependencies between the system symbol table and the other target tools are as follows:

Loader. The loader requires the system symbol table. The system symbol table does not require the presence of the loader. The target-based loader and code module management requires the system symbol table when loading code that must be linked against code already on the target.

Debugging Tools. The target-based symbolic debugging, facilities and user commands such as **i** and **tt**, rely on the system symbol table to provide information about entry points of tasks, symbolic contents of call stacks, and so on. Without the system symbol table, they can still be used but, their usefulness is greatly reduced.

Target Shell. The target shell does not strictly require the system symbol table, but its functionality is greatly limited without it. The target shell requires the system symbol table to provide the ability to run functions using their symbolic names. The target-based shell uses the system symbol table to execute shell commands, to call system routines, and to edit global variables. The target shell also includes the library **usrLib**, which contains the commands **i**, **ti**, **sp**, **period**, and **bootChange**.

wdb Agent. The wdb agent adds symbols to the system symbol table as part of the symbol synchronization with the host.

NOTE: If you choose to use both the host-resident and target-resident tools at the same time, use the synchronization method to ensure that both the host and target resident tools share the same list of symbols. The synchronization applies only to symbols and modules, not to other information such as breakpoints.

6.4.5 Synchronizing Host and Target-Resident Symbol Tables

There is another symbol table, the target server symbol table, which resides on the host and is used and maintained by the host tools. This symbol table is referred to in this chapter as the host symbol table. The host symbol table can exist even if there is no system symbol table on the target. There is an optional mechanism that relies on the **wdb** agent, which is used to synchronize the host and target symbol tables information (**INCLUDE_SYM_TBL_SYNC**). To enable this mechanism on the host, the target server must be started with the **-s** option. For more information on setting this option for the target server, see the *Tornado User's Guide*.

6.4.6 Creating User Symbol Tables

Although it is possible for a user application to manipulate symbols in the system's symbol table, this is not a recommended practice. Addition and removal of symbols to and from the symbol table is designed to be carried out only by operating system libraries. Any other use of the system symbol table may interfere with the proper operation of the operating system; and even simply introducing additional symbols could have an adverse and unpredictable effect on linking any modules that are subsequently downloaded.

Therefore, user-defined symbols should not be added programmatically to the system symbol table. Instead, when an application requires a symbol table for its own purposes, a user symbol table should be created. For more information, see the **symLib** entry in the *VxWorks API Reference*.

6.5 Show Routines

VxWorks includes system information routines which print pertinent system status on the specified object or service; however, they show only a snapshot of the system service at the time of the call and may not reflect the current state of the system. To use these routines, you must define the associated configuration macro (see the *Tornado User's Guide: Projects*). When you invoke them, their output is sent to the standard output device. Table 6-8 lists common system show routines:

Table 6-8 **Show Routines**

Call	Description	Configuration Macro
envShow( )	Displays the environment for a given task on **stdout**.	**INCLUDE_TASK_SHOW**
memPartShow( )	Shows the partition blocks and statistics.	**INCLUDE_MEM_SHOW**
memShow( )	System memory show routine.	**INCLUDE_MEM_SHOW**
moduleShow( )	Shows statistics for all loaded modules.	**INCLUDE_MODULE_MANAGER**
msgQShow( )	Message queue show utility (both POSIX and wind).	**INCLUDE_POSIX_MQ_SHOW** **INCLUDE_MSG_Q_SHOW**
semShow( )	Semaphore show utility (both POSIX and wind).	**INCLUDE_SEM_SHOW**, **INCLUDE_POSIX_SEM_SHOW**
show( )	Generic object show utility.	
stdioShow( )	Standard I/O file pointer show utility.	**INCLUDE_STDIO_SHOW**
taskSwitchHookShow( )	Shows the list of task switch routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskCreateHookShow( )	Shows the list of task create routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskDeleteHookShow( )	Shows the list of task delete routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskShow( )	Displays the contents of a task control block.	**INCLUDE_TASK_SHOW**
wdShow( )	Watchdog show utility.	**INCLUDE_WATCHDOGS_SHOW**

An alternative method of viewing system information is the Tornado browser, which can be configured to update system information periodically. For information on this tool, see the *Tornado User's Guide: Browser*.

VxWorks also includes network information show routines, which are described in the *VxWorks Network Programmer's Guide*. These routines are initialized by defining **INCLUDE_NET_SHOW** in your VxWorks configuration; see the *Tornado User's Guide: Configuration and Build*. Table 6-9 lists commonly called network show routines:

Table 6-9 **Network Show Routines**

Call	Description
ifShow( )	Display the attached network interfaces.
inetstatShow( )	Display all active connections for Internet protocol sockets.
ipstatShow( )	Display IP statistics.
netPoolShow( )	Show pool statistics.
netStackDataPoolShow( )	Show network stack data pool statistics.
netStackSysPoolShow( )	Show network stack system pool statistics.
mbufShow( )	Report **mbuf** statistics.
netShowInit( )	Initialize network show routines.
arpShow( )	Display entries in the system ARP table.
arptabShow( )	Display the known ARP entries.
routestatShow( )	Display routing statistics.
routeShow( )	Display host and network routing tables.
hostShow( )	Display the host table.
mRouteShow( )	Print the entries of the routing table.

6.6 Common Problems

This section lists frequently encountered problems that can occur when using the target tools.

Target Shell Debugging Never Hits a Breakpoint

I set a breakpoint on a function I called from the target shell, but the breakpoint is not being hit. Why not?

- **Solution**

Instead of running the function directly, use **taskSpawn( )** with the function as the entry point.

- **Explanation**

The target shell task runs with the **VX_UNBREAKABLE** option. Functions that are called directly from the target shell command prompt, are executed within the context of the target shell task. Therefore, breakpoints set within the directly called function will not be hit.

Insufficient Memory

The loader reports insufficient memory to load a module; however, checking available memory indicates the amount of available memory to be sufficient. What is happening and how do I fix it?

- **Solution**

Download the file using a different device. Loading an object module from a host file system mounted through NFS only requires enough memory for one copy of the file (plus a small amount of overhead).

- **Explanation**

The target-resident loader calls the device drivers through a VxWorks' transparent mechanism for file management, which makes calls to **open**, **close**, and **ioctl**. If you use the target-resident loader to load a module over the network (as opposed to loading from a target-system disk), the amount of memory required to load an object module depends on what kind of access is available to the remote file system over the network. This is because, depending on the device that is actually being used for the load, the calls initiate very different operations.

For some devices, the **I/O** library makes a copy of the file in target memory. Loading a file that is mounted over a device using such a driver requires enough memory to hold two copies of the file simultaneously. First, the entire file is copied to a buffer in local memory when opened. Second, the file resides in memory when it is linked to VxWorks. This copy is then used to carry out various **seek** and **read** operations. Therefore, using these drivers requires sufficient memory available to hold two copies of the file to be downloaded, as well as a small amount of memory for the overhead required or the load operation.

"Relocation Does Not Fit" Error Message

When downloading, the following types of error messages occur:

```
"Relocation does not fit in 26 bits."
```

The actual number received in the error message varies (26, or 23, or ...) depending on the architecture. What does this error mean and what should I do?

- **Solution**

Recompile the object file using **-Xcode-absolute-far** for the Diab compilers, and for GNU compilers, the appropriate "long call" option, **-m***longCallOption*.

- **Explanation**

Some architectures have instructions that use less than 32 bits to reference a nearby position in memory. Using these instructions can be more efficient than always using 32 bits to refer to nearby places in memory.

The problem arises when the compiler has produced such a reference to something that lies farther away in memory than the range that can be accessed with the reduced number of bits. For instance, if a call to **printf** is encoded with one of these instructions, the load may succeed if the object code is loaded near the kernel code, but fail if the object code is loaded farther away from the kernel image.

Missing Symbols

Symbols in code modules downloaded from the host do not appear from the target shell, and vice versa. Symbols created from the host shell are not visible from the target shell, or symbols created from the target shell are not visible from the host shell. Why is this happening, and how can I get them to appear?

- **Solution**

Check to see if the symbol synchronization is enabled for the target server as well as compiled into the image. For more information, see *6.4.5 Synchronizing Host and Target-Resident Symbol Tables*, p.268.

- **Explanation**

The symbol synchronization mechanism must be enabled separately on the host and and target.

Loader is Using Too Much Memory

Including the target loader causes the amount of available memory to be much smaller. How can I get more memory?

- **Solution**

Use the host tools (**windsh**, host loader, and so on) rather than the target tools and remove all target tools from your VxWorks image. For details, see the *Tornado User's Guide*.

- **Explanation**

Including the target loader causes the system symbol table to be included. This symbol table contains the **name**, **address**, and **type** of every global symbol in the compiled VxWorks image.

Using the target-resident loader takes additional memory away from your application—most significantly for the target-resident symbol table required by the target-resident loader.

Symbol Table Unavailable

The system symbol table failed to download onto my target. How can I use the target shell to debug the problem, since I cannot call functions by name?

- **Solution**

Use addresses of functions and data, rather than using the symbolic names. The addresses can be obtained from the VxWorks image on the host, using the **nm***arch* utility.

The following is an example from a Unix host:

```
> nmarch vxWorks | grep memShow
0018b1e8 T memShow
0018b1ac T memShowInit
```

Use this information to call the function by address from the target shell. (The parentheses are mandatory when calling by address.)

```
-> 0x0018b1e8 ()

 status   bytes    blocks   avg block  max block
------ --------- -------- ---------- ----------
current
  free   14973336     20      748666     12658120
  alloc  14201864    16163       878        -

cumulative
  alloc  21197888   142523       148        -
  value = 0 = 0x0
```

7
C++ Development

7.1 Introduction

This chapter provides information about C++ development for VxWorks using the Wind River GNU and Diab toolchains.

WARNING: GNU C++ and Diab C++ binary files are not compatible.

For documentation on using C++ with the Tornado tools, see the chapters (either in the this manual or in the *Tornado User's Guide*) that are specific to the tool.

7.2 Working with C++ under VxWorks

WARNING: Any VxWorks task that uses C++ must be spawned with the **VX_FP_TASK** option. Failure to use the **VX_FP_TASK** option can result in hard-to-debug, unpredictable floating-point register corruption at run-time. By default, tasks spawned from Tornado tools like the Wind Shell, Debugger and so on, automatically have **VX_FP_TASK** enabled.

7.2.1 Making C++ Accessible to C Code

If you want to reference a (non-overloaded, global) C++ symbol from your C code you must give it C linkage by prototyping it using **extern "C"**:

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

You can also use this syntax to make C symbols accessible to C++ code. VxWorks C symbols are automatically available to C++ because the VxWorks header files use this mechanism for declarations.

7.2.2 Adding Support Components

By default, VxWorks kernels contain only minimal C++ support. You can add C++ functionality by including any or all of the following components:

Basic Support Components

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

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

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

C++ Library Components

Several C++ library components are also available.

GNU-Specific Library Support Components

For the GNU compiler, these are:

INCLUDE_CPLUS
Includes all basic C++ run-time support in VxWorks. This enables you to download and run compiled and munched C++ modules.

INCLUDE_CPLUS_STL
Includes support for the standard template library.

INCLUDE_CPLUS_STRING
Includes the basic components of the string type library.

INCLUDE_CPLUS_IOSTREAMS
Includes the basic components of the iostream library.

INCLUDE_CPLUS_COMPLEX
Includes the basic components of the complex type library.

INCLUDE_CPLUS_IOSTREAMS_FULL
Includes the full iostream library; this component requires and automatically includes **INCLUDE_CPLUS_IOSTREAMS**.

INCLUDE_CPLUS_STRING_IO
Includes string I/O functions; this component requires and automatically includes **INCLUDE_CPLUS_STRING** and **INCLUDE_CPLUS_IOSTREAMS**.

INCLUDE_CPLUS_COMPLEX_IO
Includes I/O for complex number objects; this component requires and automatically includes **INCLUDE_CPLUS_IOSTREAMS** and **INCLUDE_CPLUS_COMPLEX**.

Diab-Specific Library Support Components

For the Diab compiler, all C++ library functionality is encapsulated in a single component:

INCLUDE_CPLUS_IOSTREAMS
Includes all library functionality.

For more information on configuring VxWorks to include or exclude these components, see the *Tornado User's Guide: Projects*.

7.2.3 The C++ Demangler

If you are using the target shell loader, you may want to include the **INCLUDE_CPLUS_DEMANGLER** component. Adding the demangler allows target shell symbol table queries to return human readable (demangled) forms of C++ symbol names. The demangler does not have to be included if you are using the host-based Tornado tools. For information about distributing the demangler, see *6.2.8 Distributing the Demangler*, p.249.

NOTE: If you want to use the target shell and C++, but do not want to include the demangler, you can exclude it from command-line-based BSP builds by defining **INCLUDE_NO_CPLUS_DEMANGLER** in **config.h**. For project builds, you simply need to remove the "C++ symbol demangler" component from your project configuration.

7.3 Initializing and Finalizing Static Objects

This section covers issues when programming with static C++ objects. These topics include an additional host processing step, and special strategies for calling static constructors and destructors.

7.3.1 Munching C++ Application Modules

Before a C++ module can be downloaded to a VxWorks target, it must undergo an additional host processing step, which by convention, is called *munching*. Munching performs the following tasks:

- Initializes support for static objects.
- Ensures that the C++ run-time support calls the correct constructors and destructors in the correct order for all static objects.
- (For GNU/ELF/DWARF only) Collapses any "linkonce" sections generated by **-fmerge-templates** into **text** or **data** (see *-fmerge-templates*, p.282).

Munching is performed after compilation and before download.

For each toolchain, the following examples compile a C++ application source file, **hello.cpp**, run munch on the **.o**, compile the generated **ctdt.c** file, and link the application with **ctdt.o** to generate a downloadable module, **hello.out**.

Using GNU

The following code includes comments and the commands to perform these steps on **hello.cpp** using the GNU toolchain:

```
# Compile
ccppc -mcpu=604 -mstrict-align  -O2 -fno-builtin -IinstallDir/target/h \
    -DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu -c hello.cpp

# Run munch
nmppc hello.o | wtxtcl installDir/host/src/hutils/munch.tcl \
    -c ppc > ctdt.c

# Compile ctdt.c file generated by munch
ccppc -mcpu=604 -mstrict-align  -fdollars-in-identifiers -O2 \
-fno-builtin -IinstallDir/target/h \
    -DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu  -c ctdt.c

# Link hello.o with ctdt.o to give a downloadable module (hello.out)
ccppc -r -nostdlib -Wl,-X -T installDir/target/h/tool/gnu/ldscripts/link.OUT \
-o hello.out hello.o ctdt.o
```

NOTE: The **-T** .../**link.OUT** option collapses any linkonce sections contained in the input files (for details, see *-fmerge-templates*, p.282). It should not be used on 68k, VxSim Solaris, or VxSim PC. Instead, you can use, for example:
cc68k -r -nostdlib -Wl,-X -o hello.out hello.o ctdt.o

Using Diab

The following code includes comments and the commands to perform these steps on **hello.cpp** using the Diab toolchain:

```
# Compile
dcc -tMCF5307FS:vxworks55 -W:c:,-Xmismatch-warning=2 \
-ew1554,1551,1552,1086,1047,1547 -Xclib-optim-off -Xansi \
-Xstrings-in-text=0 -Wa,-Xsemi-is-newline -ei1516,1643,1604 \
-Xlocal-data-area-static-only -ew1554 -XO -Xsize-opt -IinstallDir/target/h \
    -DCPU=MCF5200 -DTOOL_FAMILY=diab -DTOOL=diab -c hello.cpp

# Run munch
nmcf hello.o | wtxtcl <italic>installDir</italic>/host/src/hutils/munch.tcl \
-c cf > ctdt.c
```

```
# Compile ctdt.c file generated by munch
dcc -tMCF5307FS:vxworks55 -Xdollar-in-ident -XO -Xsize-opt -Xlint \
-I<italic>installDir</italic>/target/h \
    -DCPU=MCF5200 -DTOOL_FAMILY=diab -DTOOL=diab -c ctdt.c

# Link hello.o with ctdt.o to give a downloadable module (hello.out)
dld -tMCF5307FS:vxworks55 -X -r -r4 -o hello.out hello.o ctdt.o
```

Using a Generic Rule

If you use the VxWorks Makefile definitions, you can write a simple munching rule which (with appropriate definitions of CPU and TOOL) works across all architectures for both GNU and Diab toolchains.

```
CPU     = PPC604
TOOL    = gnu

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp

default : hello.out

%.o : %.cpp
    $(CXX) $(C++FLAGS) -c $<

%.out : %.o
    $(NM) $*.o | $(MUNCH) > ctdt.c
    $(CC) $(CFLAGS) $(OPTION_DOLLAR_SYMBOLS) -c ctdt.c
    $(LD_PARTIAL) $(LD_PARTIAL_LAST_FLAGS) -o $@ $*.o ctdt.o
```

After munching, downloading, and linking, the static constructors and destructors are called. This step is described next.

7.3.2 Calling Static Constructors and Destructors Interactively

VxWorks provides two strategies for calling static constructors and destructors. These are described as follows:

automatic invocation
This is the default strategy. Static constructors are executed just after the module is downloaded to the target and before the module loader returns to its caller. Static destructors are executed just prior to unloading the module.

manual invocation

Requires the user to call static constructors manually, after downloading the module, but before running the application. Requires the user to call static destructors manually, after the task finishes running, but before unloading the module. Static constructors are called by invoking **cplusCtors( )**. Static destructors are called by invoking **cplusDtors( )**. These routines take a module as an argument; thus, static constructors and destructors are called explicitly on a module-by-module basis. However, you can also invoke all currently-loaded static constructors or destructors by calling these routines with no argument.

CAUTION: When using the manual invocation method, constructors for each module must only be run once.

You can change the strategy for calling static constructors and destructors by using **cplusXtorSet( )**. To report on the current strategy, call **cplusStratShow( )**.

For more information on the routines mentioned in this section, see the API entries in the online reference manuals.

7.4 Programming with GNU C++

The GNU compilers provided with the Tornado IDE support most of the language features described in the *ANSI C++ Standard*. In particular, they provide support for template instantiation, exception handling, run-time type information, and namespaces. Details regarding these features, as supported by the GNU toolchain, are discussed in the following sections.

For complete documentation on the GNU compiler and on the associated tools, see the *GNU ToolKit User's Guide*.

7.4.1 Template Instantiation

The GNU C++ toolchain supports three distinct template instantiation strategies. These are described below.

-fimplicit-templates

This is the option for implicit instantiation. Using this strategy, the code for each template gets emitted in every module that needs it. For this to work the body of a template must be available in each module that uses it. Typically this is done by including template function bodies along with their declarations in a header file. The advantage of implicit instantiation is that it is simple and it is used by default in VxWorks makefiles. The disadvantage is that it may lead to code duplication and larger application size.

-fmerge-templates

This is like **-fimplicit-templates** option, except that 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. Thus **-fmerge-templates** avoids the code bloat associated with **-fimplicit-templates**.

NOTE: This flag is only supported on ELF/DWARF targets; it is accepted, but ignored by the 68k and simulator compilers (**cc68k**, **ccsimso**, and **ccsimpc**).

CAUTION: The VxWorks dynamic loader does not support "linkonce" sections directly. Instead, the linkonce sections must be merged and collapsed into standard **text** and **data** sections before loading. This is done with a special link step described in *7.3.1 Munching C++ Application Modules*, p.278.

-fno-implicit-templates

This is the option for explicit instantiation. Using this strategy explicitly instantiates any templates that you require. The advantage of explicit instantiation is that it allows you the most control over where templates get instantiated and avoids code bloat. This disadvantage is that you need to explicitly instantiate each template.

-frepo

This is the option for the third approach. This strategy works by manipulating a database of template instances for each module. The compiler generates a **.rpo** file, for each corresponding object file, which lists all template instantiations that are used, and could be instantiated, in that object file. The link wrapper, **collect2**, then updates the **.rpo** files to tell the compiler where to place those instantiations, and rebuilds any affected object files. The link-time overhead is negligible after the first pass, as the compiler continues to place the instantiations in the same files. The advantage of this approach is that it combines the simplicity of implicit instantiation with the smaller footprint obtained by instantiating templates by hand.

Procedure

The header file for a template must contain the template body. If template bodies are currently stored in **.cpp** files, the line **#include** *theTemplate***.cpp** must be added to *theTemplate***.h**.

A complete build with the **-frepo** option is required to create the **.rpo** files that tell the compiler which templates to instantiate. The link step should be invoked using **cc***arch* rather than **ld***arch*.

Subsequently, individual modules can be compiled as usual (but with the **-frepo** option and no other template flags).

When a new template instance is required, the relevant part of the project must be rebuilt to update the **.rpo** files.

Loading Order

The Tornado tools' dynamic linking requires the module containing a symbol definition to be downloaded before a module that references it. When using the **-frepo** option, it may not be clear which module contains the definition. Thus, you should also prelink them and download the linked object.

Example

This example uses a standard VxWorks BSP makefile; for concreteness, it uses a 68K target.

Example 7-1 **Sample Makefile**

```
make PairA.o PairB.o ADDED_C++FLAGS=-frepo

/* dummy link step to instantiate templates */
cc68k -r -o Pair PairA.o PairB.o

//Pair.h

template <class T> class Pair
```

```
{
public:
    Pair (T _x, T _y);
    T Sum ();

protected:
    T x, y;
};

template <class T>
Pair<T>::Pair (T _x, T _y) : x (_x), y(_y)
{
}

template <class T>
T Pair<T>::Sum ()
{
    return x + y;
}

// PairA.cpp
#include "Pair.h"

int Add (int x, int y)
{
    Pair <int> Two (x, y);
    return Two.Sum ();
}

// PairB.cpp
#include "Pair.h"

int Double (int x)
{
    Pair <int> Two (x, x);
    return Two.Sum ();
}
```

7.4.2 Exception Handling

The GNU C++ compiler supports multi-thread safe exception handling by default. To turn **off** support for exception handling, use the **-fno-exceptions** compiler flag.

The following topics regarding exception handling are discussed throughout this section:

- the affect on code written for the pre-exception model
- the overhead for using exception handling
- how to install your own termination handler

Using the Pre-Exception Model

You can still write code according to the pre-exception model of C++ compilation. For example, your calls to **new** can check the returned pointer for a failure value of zero. However, if you are concerned that the exception handling enhancements in this release will not compile your code correctly, follow these simple rules:

- Use **new** (**nothrow**).
- Do not explicitly turn on exceptions in your iostreams objects.
- Do not use string objects or wrap them in "try { } catch (...) { }" blocks.

These rules derive from the following observations:

- GNU iostreams does not **throw** unless **IO_THROW** is defined when the library is built, and exceptions are explicitly enabled for the particular iostreams object in use. The default is no exceptions. Exceptions have to be explicitly turned on for each **iostate** flag that wants to throw.
- The Standard Template library (STL) does not throw except in some methods in the **basic_string** class (of which "string" is a specialization).

Exception Handling Overhead

Exception handling creates a high overhead. To support the destruction of automatic objects during stack-unwinding, for example, the compiler must insert additional code for any function that creates an automatic (stack based) object with a destructor.

As an example of this additional code, the following demonstrates using exception handling on a PowerPC 604 target (**mv2604** BSP). The counts are measured in executed instructions. The instructions listed below are those that need to be added for exception handling.

- In order to execute a "**throw 1**" and the associated "**catch (...)**", 1235 instructions are executed.
- In order to register and de-register automatic variables and temporary objects with destructors, 14 instructions are executed.
- For each non-inlined function that uses exception handling, 29 instructions are executed to perform the exception-handling setup.
- Upon encountering the first exception-handling construct (**try**, **catch**, **throw**, or registration of an **auto** variable or temporary), 947 instructions are executed.

The example code follows:

```
                      first time   normal case
void test()  {          // 3+29         3+29
    throw 1;            // 1235         1235     total time to printf
}

void doit() {           // 3+29+947     3+29
    try {               // 22           22
    test();             // 1            1
    } catch (...) {
    printf("Hi\n");
    }
}

struct A { ~A( ) { } };

void local_var ( ) {    //              3+29
    A a;                //              14
}                       //              4
```

You can turn **off** exception handling by using the **-fno-exceptions** option. Doing so removes all exception handling overhead.

Unhandled Exceptions

As required by the *ANSI C++ Standard*, an unhandled exception ultimately calls **terminate( )**. The default behavior of this routine is to suspend the offending task and to send a warning message to the console. You can install your own termination handler by calling **set_terminate( )**, which is defined in the header file **exception**.

7.4.3 Run-Time Type Information

The GNU C++ compiler supports the Run-time Type Information (RTTI) feature. This feature is turned **on** by default and adds a small overhead to any C++ program containing classes with virtual functions. If you do not need this feature, you can turn it **off** using **-fno-rtti**.

7.4.4 Namespaces

The GNU C++ compiler supports namespaces. You can use namespaces for your own code, according to the C++ standard.

The final version of 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. For the **std** namespace, the GNU C++ compiler accepts the new format, but does not require it. This is because GNU C++ puts the **std** namespace into the global namespace.

This means that the current GNU C++ compiler is transitional, compiling both legacy code and new code written according to the standard. However, remember when coding under the new syntax that identifiers in the **std** namespace will be global; and therefore, they must be unique for the global namespace.

As an example, the following code is technically invalid under the latest standard, but will compile under the current GNU C++ compiler:

```
#include <iostream.h>
int main()
    {
    cout << "Hello, world!" << endl;
    }
```

The following three examples show how the C++ standard would now represent this code. These examples will also compile under the current 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;
    }
```

Note that the **using** directives is accepted, but not required by GNU C++.

The same applies to standard C code. For example, both of these code examples will compile:

```
#include <stdio.h>

void main()
    {
    int i = 10;
    printf("%d", &i");
    }
```

or

```
#include <cstdio>

void main()
    {
    int i = 10;
    std::printf("%d", &i");
    }
```

7.5 Programming with Diab C++

The Diab C++ compilers provided with the Tornado IDE use the Edison Design Group (EDG) C++ front end. The Diab C++ compiler fully supports the ANSI C++ Standard, except for the limitations mentioned in *7.5.1 Template Instantiation*, p. 288.

The following sections briefly describe template instantiation, exception handling, and run-time type information as supported by the Diab compiler. For complete documentation on the Diab compiler and associated tools, see the *Diab C/C++ Compiler User's Guide*.

7.5.1 Template Instantiation

Function and class templates are implemented according to the standard, except that:

- The **export** keyword is not implemented for templates.
- A partial specialization of a class member template cannot be added outside of the class definition.

There are two ways to control instantiation of templates. By default, templates are instantiated *implicitly*--that is, they are instantiated by the compiler whenever a template is used. 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).
```

The Diab C++ options summarized below control multiple instantiation of templates. For more information about these options, see the *Diab C/C++ Compiler User's Guide*.

-Ximplicit-templates

Instantiate each template wherever used. This is the default.

-Ximplicit-templates-off

Instantiate templates only when explicitly instantiated in code.

-Xcomdat

When templates are instantiated implicitly, mark each generated **code** or **data** section as **comdat**. The linker collapses identical instances marked as such, into a single instance in memory.

VxWorks cannot load object files that contain **comdat**s, however, you can use **-Xcomdat-on** with the Diab linker, and load the resulting executable.

-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.This option is the default.

7.5.2 Exception Handling

The Diab C++ compiler supports thread safe exception handling by default. To turn off support for exception handling, use the **-Xexceptions-off** compiler flag.

The Diab exception handling model is table driven and requires little run-time overhead if a given exception is not thrown. Exception handling does, however, involve a size increase.

7.5.3 Run-Time Type Information

The Diab C++ compiler supports C++ Run-time Type Information (RTTI). This language feature is enabled by default and can be disabled with the **-Xrtti-off** compiler flag.

7.6 Using C++ Libraries

The iostream libraries, and the string and complex number classes, are provided in VxWorks components and are configurable with either toolchain. These libraries are described below.

String and Complex Number Classes

These classes are part of the new Standard C++ library. For the Diab toolchain these classes are provided entirely in header files. For the GNU toolchain they can be configured into a VxWorks system with **INCLUDE_CPLUS_STRING** and **INCLUDE_CPLUS_COMPLEX**. You may optionally include I/O facilities for these classes with **INCLUDE_CPLUS_STRING_IO** and **INCLUDE_CPLUS_COMPLEX_IO**.

iostreams Library

This library is configured into your VxWorks system with the **INCLUDE_CPLUS_IOSTREAMS** component.

For the GNU toolchain, the iostreams library header files reside in *installDir*/**host/***hostType*/**include/g++-3** directory. For the Diab toolchain, the iostreams library header files reside in the *installDir*/**host/diab/include/cpp** directory.

WARNING: Each compiler automatically includes the directories containing the header files for these libraries. You should never explicitly add those directories to your compiler include path. If you get warnings about missing headers, it probably means you are using the **-nostdinc** flag. This flag should never be used with this release of Tornado.

To use this library, include one or more of the header files after the **vxWorks.h** header in the appropriate modules of your application. The most frequently used header file is **iostream.h**, but others are available; see a C++ reference such as Stroustrup for information.

The standard iostreams objects (**cin**, **cout**, **cerr**, and **clog**) are global; that is, they are not **private** to any given task. They are correctly initialized regardless of the number of tasks or modules that reference them and they can safely be used across multiple tasks that have the same definitions of **stdin**, **stdout**, and **stderr**. However they cannot safely be used when different tasks have different standard I/O file descriptors; in such cases, the responsibility for mutual exclusion rests with the application.

The effect of **private** standard iostreams objects can be simulated by creating a new iostreams object of the same class as the standard iostreams object (for example, **cin** is an **istream_withassign**), and assigning to it a new **filebuf** object tied to the appropriate file descriptor. The new **filebuf** and iostreams objects are **private** to the calling task, ensuring that no other task can accidentally corrupt them.

```
ostream my_out (new filebuf (1));         /* 1 == STDOUT */
istream my_in (new filebuf (0), &my_out); /* 0 == STDIN;
                                           * TIE to my_out */
```

For complete details on the iostreams library, see the *GNU ToolKit User's Guide*.

Standard Template Library (STL)

For both toolchains, the Standard Template library consists entirely of a set of header files. There is no special run-time component that must be included in a VxWorks system.

STL for GNU

The GNU STL port for VxWorks is thread safe at the class level. This means that the client has to provide explicit locking, if two tasks want to use the same container object (a semaphore can be used to do so; see *2.3.3 Semaphores*, p. 34). However, two different objects of the same STL container class can be accessed concurrently.

The C++ Standard Template Library can be used in client code compiled with exception handling turned off. In our implementation this has the following semantics:

(1) For all checks that could reasonably be made by the caller, such as bounds checking, no action is taken where an exception would have been thrown. With optimization on, this is equivalent to removing the check.

(2) For memory exhaustion where **bad_alloc** would have been thrown, now the following message is logged (if logging is included):

```
"STL Memory allocation failed and exceptions disabled -calling terminate"
```

and the task calls **terminate**. This behavior applies only to the default allocators; you are free to define custom allocators with a different behavior.

STL for Diab

The Diab C++ library is fully compliant with the ANSI C++ Standard with the following minor exception: it does not have full **wchar** support. The Diab STL component is thread safe.

7.7 Running the Example Demo

The **factory** demo example demonstrates various C++ features, including the Standard Template Library, user-defined templates, run-time type information, and exception handling. This demo is located in *installDir*/**target/src/demo/cplusplus/factory**.

To create, compile, build, and run this demo program you can use either the Tornado project tool, documented in the *Tornado User's Guide: Projects*, or the command-line, as shown below.

For the **factory** demo, your kernel must include the following components:

- **INCLUDE_CPLUS**
- **INCLUDE_CPLUS_LANG**
- **INCLUDE_CPLUS_IOSTREAMS**

In addition, for GNU only, include the following components:

- **INCLUDE_CPLUS_STRING**
- **INCLUDE_CPLUS_STRING_IO**

To add components from the Tornado IDE, see the *Tornado User's Guide: Projects*.

To build **factory** from the command line, simply copy the **factory** sources to the BSP directory, as shown below:

```
cd installDir/target/config/bspDir
cp installDir/target/src/demo/cplusplus/factory/factory.* .
```

Then, to build a bootable image containing the **factory** example, run make as shown below:

```
make ADDED_MODULES=factory.o
```

and boot the target.

To build a downloadable image containing the **factory** example, run make as shown below:

```
make factory.out
```

Then, from the WindSh, load the **factory** module, as shown below:

```
ld < factory.out
```

Finally, to run the **factory** demo example, type at the shell:

```
-> testFactory
```

Full documentation on what you should expect to see is provided in the source code comments for the demo program.

8

Flash Memory Block Device Driver

Optional Component TrueFFS

8.1 Introduction

TrueFFS for Tornado is an optional product that provides a block device interface to a wide variety of flash memory devices. TrueFFS is a VxWorks-compatible implementation of M-Systems FLite, version 2.0. This system is reentrant, thread-safe, and supported on all CPU architectures that host VxWorks.

This chapter begins with a brief introduction to flash memory, followed by a step-by-step outline of the procedure for building a system with TrueFFS. Then, the main part of the chapter describes these steps in detail, including sections devoted to writing your own socket driver and MTD components. The chapter ends with a description of the functionality of flash memory block devices.

NOTE: This version of the TrueFFS product is a block device driver to VxWorks that, although intended to be file system neutral, is guaranteed to work only with the MS-DOS compatible file system offered with this product.

8.1.1 Choosing TrueFFS as a Medium

TrueFFS applications can read and write from flash memory just as they would from an MS-DOS file system resident on a magnetic-medium mechanical disk drive. However, the underlying storage media are radically different. While these differences are completely transparent to the high-level developer, it is critical that you be aware of them when designing an embedded system.

Flash memory stores data indefinitely, even while unpowered. The physical components of flash memory are solid-state devices[1] that consume little energy and leave a small foot print. Thus, flash memory is ideal for mobile devices, for hand-held devices, and for embedded systems that require reliable, non-volatile environments that are too harsh for mechanical disks.

However, flash does have a limited life, due to the finite number of erase cycles; and, TrueFFS only supports flash devices that are symmetrically blocked. In addition, some features common to block device drivers for magnetic media are not available with flash memory. Unequal read and write time is a typical characteristic of flash memory, in which reads are always faster than writes. For more information, see *8.13 Flash Memory Functionality*, p.338. Also, TrueFFS does not support **ioctl**.

The unique qualities that distinguish flash memory from magnetic-media disk drives make it an ideal choice for some types of applications, yet impractical for others.

NOTE: Although you can write in any size chunks of memory, ranging from bytes to words and double words, you can only erase in blocks. The best approach to extending the life of the flash is to ensure that all blocks wear evenly. For more information, see *8.13 Flash Memory Functionality*, p.338.

NOTE: The TrueFFS optional product does have not support for partition tables.

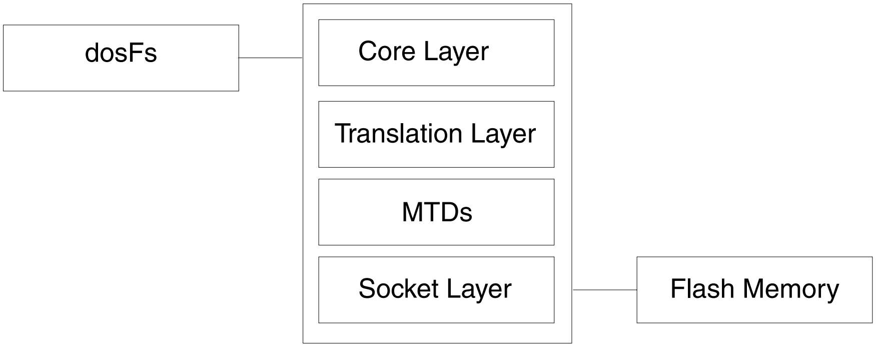
⚠ **CAUTION:** VxWorks favors task priorities over I/O sequence when servicing I/O requests. Thus, if a low priority task and a high priority task request an I/O service while the resource is busy, the high priority task gets the resource when it becomes available—even if the low priority got its request in before the high priority task. To VxWorks, a flash device is just another resource.

8.1.2 TrueFFS Layers

TrueFFS is comprised of a core layer and three functional layers–the translation layer, the Memory Technology Driver (MTD) layer, and the socket layer–as illustrated in Figure 8-1. The three functional layers are provided in source code form or in binary form, or in both, as noted below. For more detailed information about the functionality of these layers, see *8.13 Flash Memory Functionality*, p.338.

[1] Meaning that they have no moving parts.

Figure 8-1 **TrueFFS is a Layered Product**



- **Core Layer** This layer connects other layers to each other. It also channels work to the other layers and handles global issues, such as "backgrounding", garbage collection, timers, and other system resources. The core layer is provided in binary form only.

- **Translation Layer** This layer maintains the map that associates the file system's view of the storage medium with the erase blocks in flash. The Block Allocation Map is the basic building block for implementing wear leveling and error recovery. The translation layer is media specific (NOR or SSFDC). The translation layer is provided in binary form only.

- **MTD Layer** The MTD implements the low-level programming (map, read, write, and erase) of flash medium. MTDs are provided in both source and binary form.

- **Socket Layer** The socket layer provides the interface between TrueFFS and the board hardware, providing board-specific hardware access routines. It is responsible for power management, card detection, window management, and socket registration. The socket drivers are provided in source code only.

8.2 Building Systems with TrueFFS

This section presents a high-level overview of the development process, outlining the steps required to configure and build a VxWorks system that supports TrueFFS. This process applies to VxWorks systems that can be used with either bootable or downloadable applications.

Step 1: **Select an MTD Component**

Choose an MTD, appropriate to your hardware, from those provided with the TrueFFS product. You may prefer to (or, in rare cases, need to) write your own. For details, see *8.3 Selecting an MTD Component*, p.299.

Step 2: **Identify the Socket Driver**

Ensure that you have a working socket driver. The socket driver is a source code component, implemented in the file **sysTffs.c**. For some BSPs, the socket driver is fully defined and located in the BSP directory. If it is not, you can port a generic file containing skeleton code to your hardware. For details, see *8.4 Identifying the Socket Driver*, p.300.

Step 3: **Configure the System**

Configure your system for TrueFFS by adding the appropriate components. Minimum support requires components for dosFs and the four TrueFFS layers. For details, see *8.5 Configuring and Building the Project*, p.300.

NOTE: Component descriptions use the abbreviation **TFFS**, rather than **TRUEFFS**, as might be expected.

Step 4: **Build the Project**

Before you build the system, the binaries for the MTDs need to be up to date and the socket driver file located in the BSP directory must be a working version. For details, see *8.5.7 Building the System Project*, p.305.

Step 5: **Boot the Target and Format the Drives**

Next, boot the target. Then, from the shell, format the drives. For details, see *8.6 Formatting the Device*, p.306.

NOTE: To preserve a region of flash for boot code, see *8.7 Creating a Region for Writing a Boot Image*, p.309.

Step 6: **Mount the Drive**

Mount the VxWorks DOS file system on a TrueFFS flash drive. For details, see *8.8 Mounting the Drive*, p.312.

Step 7: **Test the Drive**

Test your drive(s).

One way to do this is to perform a quick sanity check by copying a text file from the host (or from another type of storage medium) to the flash file system on the target; then copy the file to the console or to a temporary file for comparison, and verify the content. The following example is run from the shell as a sanity check:

```
%->@copy "host:/home/panloki/.cshrc" "/flashDrive0/myCshrc"
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
%->@copy "/flashDrive0/myCshrc"
...
...
...
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
```

NOTE: The copy command requires the appropriate configuration of dosFs support components. For details, see *Optional dosFs Components*, p.197.

8.3 Selecting an MTD Component

The directory *installDir*/**target/src/drv/tffs** contains the source code for the following types of MTD components:

- MTDs that work with several of the devices provided by Intel, AMD, Fujitsu, and Sharp.
- Two generic MTDs that can be used for devices complying with CFI.

To better support the out-of-box experience, these MTDs attempt to cover the widest possible range of devices (in their class) and of bus architectures. Consequently, the drivers are bulky and slow in comparison to drivers written specifically to address the runtime environment that you want to target. If the performance and size of the drivers provided do not match your requirements, you can modify them to better suit your needs. The section *8.12 Writing MTD Components*, p.327 has been provided exclusively to address this issue.

For a complete list of these MTDs, see *8.11 Using the MTD-Supported Flash Devices*, p.323. In addition, section *8.11.1 Supporting the Common Flash Interface (CFI)*, p.323 describes the CFI MTDs in detail. Evaluate whether any of these drivers support the device that you intend to use for TrueFFS. Devices are usually identified by their JEDEC IDs. If you find an MTD appropriate to your flash device, you can use that MTD. These drivers are also provided in binary form; so you do not need to compile the MTD source code unless you have modified it.

NOTE: For a list of the MTD components and details about adding the MTD component to your system project, see *8.5.4 Including the MTD Component*, p.303.

8.4 Identifying the Socket Driver

The socket driver that you include in your system must be appropriate for your BSP. Some BSPs include socket drivers, others do not. The socket driver file is **sysTffs.c** and, if it exists, it is located in your BSP directory. If your BSP provides a socket driver, you can use it.

If your BSP does not provide this file, follow the procedure described in *8.10 Writing Socket Drivers*, p.314, which explains how to port a stub version to your hardware.

In either case, the build process requires that a working socket driver (**sysTffs.c**) be located in the BSP directory. For more information, see *8.5.6 Adding the Socket Driver*, p.305.

8.5 Configuring and Building the Project

VxWorks systems configured with TrueFFS include:

- configuration to fully support the dosFs file system
- a core TrueFFS component, **INCLUDE_TFFS**
- at least one software module from each of the three TrueFFS layers

You can configure and build your system either from the command line or by using the Tornado IDE project facility. When choosing a method for configuring and building systems with TrueFFS, consider the following criteria:

- Configuring and building from the command line involves editing text files that contain component listings and parameters for initialization, and calling the **make** utility to build a system image for you. This process, while possibly faster than using the project facility, requires that you provide the list of dependent components accurately.
- Configuring and building through the Tornado project facility provides an easy and accurate method of adding needed components, while the build process can take longer than when using the command line.

For both configuration and build methods, special consideration must be given to cases where either the socket driver or the MTD, or both, are not provided. The drivers need to be registered and MTDs need appropriate component descriptions. For more information, see *8.10 Writing Socket Drivers*, p.314, and *8.12.4 Defining Your MTD as a Component*, p.335.

For general information on configuration procedures, see the *Tornado User's Guide: Configuration and Build* and the *Tornado User's Guide: Projects*.

NOTE: Included with TrueFFS for Tornado are sources for several MTDs and socket drivers. The MTDs are in **target/src/drv/tffs**. The socket drivers are defined in the **sysTffs.c** files provided in the **target/config/***bspname* directory for each BSP that supports TrueFFS.

8.5.1 Including File System Components

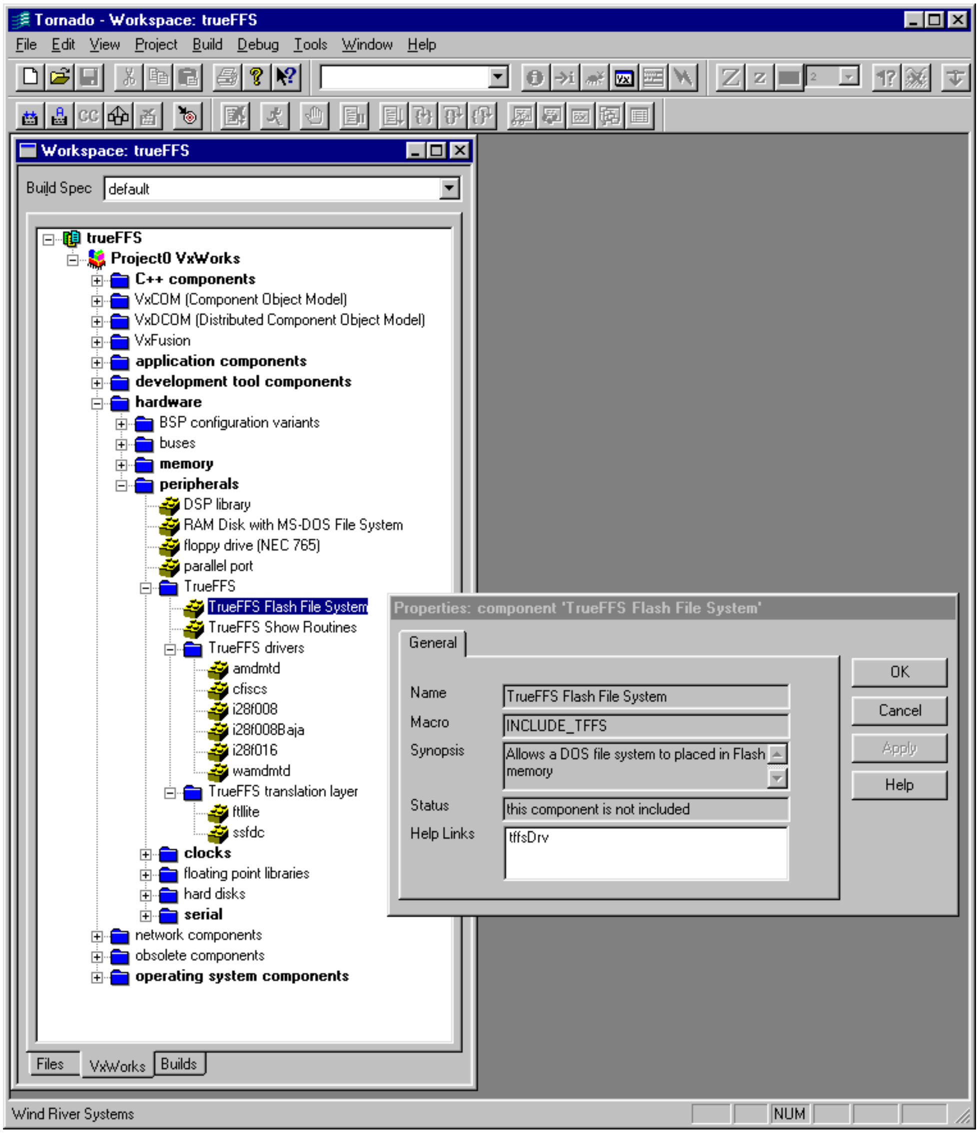
A system configuration with TrueFFS is essentially meaningless without the VxWorks compatible file system, MS-DOS. Therefore, both dosFs support, and all components that it depends upon, need to be included in your TrueFFS system. For information on this file system and support components, see *5.2.2 Configuring Your System*, p.197.

In addition, there are other file system components that are not required, but which may be useful. These components add support for the basic functionality that one needs to use a file system, such as the commands **ls**, **cd**, **copy**, and so.

8.5.2 Including the Core Component

All systems must include the TrueFFS core component, **INCLUDE_TFFS**.

Figure 8-2 **Adding TrueFFS Components from the Project Facility**



Name: TrueFFS Flash File System

Macro: INCLUDE_TFFS

Synopsis: Allows a DOS file system to placed in Flash memory

Status: this component is not included

Help Links: tffsDrv

Defining this component triggers the correct sequence of events, at boot time, for initializing this product. It also ensures that the socket driver is included in your project (see *8.5.6 Adding the Socket Driver*, p.305).

You can include this component using the project facility, as shown in Figure 8-2, or by defining **INCLUDE_TFFS** in **config.h**.

8.5.3 Including Utility Components

This section describes TrueFFS utility components, their purpose, and their default configuration options. You do not need to modify the default configuration for these components in order to create a functioning TrueFFS system.

INCLUDE_TFFS_SHOW

Including this component adds two TrueFFS configuration display utilities, **tffsShow( )** and **tffsShowAll( )**. The **tffsShow( )** routine prints device information for a specified socket interface. It is particularly useful when trying to determine the number of erase units required to write a boot image. The **tffsShowAll( )** routine provides the same information for all socket interfaces registered with VxWorks. The **tffsShowAll( )** routine can be used from the shell to list the drives in the system. The drives are listed in the order in which they were registered. This component is not included by default. You can include it from the project facility or by defining **INCLUDE_TFFS_SHOW** in **config.h**.

NOTE: INCLUDE_TFFS_BOOT_IMAGE is defined by default in the socket driver, **sysTffs.c**. It is used for configuring flash-resident boot images. Defining this constant automatically includes **tffsBootImagePut( )** in your **sysTffs.o**. This routine is used to write the boot image to flash memory (see *8.7.3 Writing the Boot Image to Flash*, p.311).

8.5.4 Including the MTD Component

Add the MTD component appropriate to your flash device (*8.3 Selecting an MTD Component*, p.299) to your system project. If you have written your own MTD, see *8.12.4 Defining Your MTD as a Component*, p.335 to ensure that it is defined correctly for inclusion. You can add the MTD component either through the project facility or, for a command-line build, by defining it in the socket driver file (see *8.5.7 Building the System Project*, p.305). Whichever method is used, it must be the same for configuring the MTD component and building the project.

The MTD components provided by TrueFFS, for flash devices from Intel, AMD, Fujitsu, and Sharp, are listed below:

INCLUDE_MTD_CFISCS
CFI/SCS device; for details, see *CFI/SCS Flash Support*, p.324.

INCLUDE_MTD_CFIAMD
CFI-compliant AMD and Fujitsu devices; for details, see *AMD/Fujitsu CFI Flash Support*, p.325.

INCLUDE_MTD_I28F016
Intel 28f016; for details, see *Intel 28F016 Flash Support*, p.325.

INCLUDE_MTD_I28F008
Intel 28f008; for details, see *Intel 28F008 Flash Support*, p.326.

INCLUDE_MTD_AMD
AMD, Fujitsu: 29F0{40,80,16} 8-bit devices; for details, see *AMD/Fujitsu Flash Support*, p.326.

INCLUDE_MTD_WAMD
AMD, Fujitsu 29F0{40,80,16} 16-bit devices.

INCLUDE_MTD_I28F008_BAJA
Intel 28f008 on the Heurikon Baja 4000.

MTDs defined in the component descriptor file (that is, included through the project facility) usually make it explicit to require the translation layer. However, if you are building from the command-line, or writing your own MTD, you may need to explicitly include the translation layer.

NOTE: Although components can also be defined in **config.h**, this is not recommended for the MTD component because it can conflict with the project facility configurations.

8.5.5 Including the Translation Layer

The translation layer is selected according to the technology used by your flash medium. The two types of flash are NOR and NAND. This product only supports NAND devices that conform to the SSFDC specification. For more information, see *8.11.3 Obtaining Disk On Chip Support*, p.327.

The translation layer is provided in binary form only. The translation layer components that are provided are listed below.

INCLUDE_TL_FTL
The translation layer for NOR flash devices. If you can execute code in flash, your device uses NOR logic.

INCLUDE_TL_SSFDC
The translation layer for devices that conform to Toshiba's Solid State Floppy Disk Controller Specifications. TrueFFS supports NAND devices that only comply with the SSFDC specification.

The component descriptor files (for the project facility) specify the dependency between the translation layers and the MTDs; therefore, when configuring through the project facility, you do not need to explicitly select a translation layer. The build process handles it for you.

If you are not using the project facility, you are responsible for selecting the correct translation layer. As with the MTD, when configuring and building from the command line, you define the translation layer in **sysTffs.c** within the conditional clause **#ifndef PROJECT_BUILD**. For details, see *nConditional Compilation*, p.305.

For more information, see *8.12.4 Defining Your MTD as a Component*, p.335.

8.5.6 Adding the Socket Driver

To include the socket driver in your project, a working version of the socket driver, **sysTffs.c**, must be located in you BSP directory.

Inclusion of the socket driver is relatively automatic. By including the core TrueFFS component, **INCLUDE_TFFS**, into your project, the build process checks for a socket driver file, **sysTffs.c**, in the BSP directory and includes that file in the system project.

If your BSP does not provide a socket driver, follow the procedure described in *8.10 Writing Socket Drivers*, p.314.

8.5.7 Building the System Project

Build the system project, from either the command line or the Tornado IDE. When building the system project, consider the following two issues.

- **Conditional Compilation** The file **sysTffs.c** defines constants within a conditional clause **#ifndef PROJECT_BUILD**. By default, these constants include definitions for all the MTD components provided with the TrueFFS product. This **PROJECT_BUILD** clause conditionally *includes* all of these constants for a command-line build (adds them to **sysTffs.o**), and *excludes* them for a project facility build (because you include them from the GUI). Therefore, the same method must be used to both configure the MTD and the translation layer components, and to build the project.

If you are building from the command line, and want to save on memory space, you can undefine the constants for any MTDs that your hardware does not support.

- **Up-to-Date Components** Before you build the project, the binaries for the MTDs must be up to date, and the **sysTffs.c** that is present in the BSP directory must be a working version.

  MTDs are provided in source and binary form. When writing your own MTD, rebuilding the directory is the recommended way to transform the source to binary. This way the binary is placed in the right location and added to the appropriate library. If any files in the group *installDir***/target/src/drv/tffs/*.c** are newer than corresponding files in the library *installDir***/target/lib/***archFamily***/***arch***/common/libtffs.a**, then rebuild them before building the project.

8.6 Formatting the Device

Boot the system. After the system boots and registers the socket driver(s), bring up the shell. From the shell, run **tffsDevFormat( )** to format the flash memory for use with TrueFFS. This routine, as defined in **tffsDrv.h**, takes two arguments, a drive number and a format argument:

```
tffsDevFormat (int tffsDriveNo, int formatArg);
```

NOTE: You can format the flash medium even though there is not yet a block device driver associated with the flash.

CAUTION: Running **tffsDevFormat( )** on a device that is sharing boot code with the file system will leave the board without boot code at the end of it. The board then becomes unusable until some alternative method is provided for reflashing the lost image. Once you do that, the file system that you created by formatting is destroyed.

8.6.1 Specifying the Drive Number

The first argument, **tffsDriveNo**, is the drive number (socket driver number). The drive number identifies the flash medium to be formatted and is determined by the order in which the socket drivers were registered. Most common systems have a single flash drive, but TrueFFS supports up to five. Drive numbers are assigned to the flash devices on your target hardware by the order in which the socket drivers are registered in **sysTffsInit( )** during boot. The first to be registered is drive 0, the second is drive 1, and so on up to 4. Therefore, the socket registration process determines the drive number. (Details of this process are described in *Socket Registration*, p.320.) You use this number to specify the drive when you format it.

8.6.2 Formatting the Device

The second argument, **formatArg**, is a pointer to a **tffsDevFormatParams** structure (cast to an **int**). This structure describes how the volume should be formatted. The **tffsDevFormatParams** structure is defined in *installDir***target\h\tffs\tffsDrv.h** as:

```
typedef struct
    {
     tffsFormatParams  formatParams;
     unsigned          formatFlags;
    }tffsDevFormatParams;
```

The first member, **formatParams**, is of type **tffsFormatParams**. The second member, **formatFlags**, is an **unsigned int**.

TFFS_STD_FORMAT_PARAMS Macro

To facilitate calling **tffsDevFormat(**) from a target shell, you can simply pass zero (or a NULL pointer) for the second argument, **formatArg**. Doing so will use a macro, which defines default values for the **tffsDevFormatParams** structure. The macro, **TFFS_STD_FORMAT_PARAMS**, defines the default values used in formatting a flash disk device. This macro, **TFFS_STD_FORMAT_PARAMS**, is defined in **tffsDrv.h** as:

```
#define TFFS_STD_FORMAT_PARAMS {{0, 99, 1, 0x100001, NULL, {0,0,0,0},
NULL, 2, 0, NULL}, FTL_FORMAT_IF_NEEDED}
```

If the second argument, **formatArg**, is zero, **tffsDevFormat( )** uses the default values from this macro.

The macro passes values for both the first and second members of the **tffsDevFormatParams** structure. These are:

```
formatParams = {0, 99, 1, 0x100001, NULL, {0,0,0,0}, NULL, 2, 0, NULL}
formatFlags = FTL_FORMAT_IF_NEEDED
```

The meaning of these default values, and other possible arguments for the members of this structure, are described below.

formatParams Member

The **formatParams** member is of the type **tffsFormatParams**. Both this structure, and the default values used by the **TFFS_STD_FORMAT_PARAMS** macro, are defined in *installDir***\target\h\tffs\tffsDrv.h**.

If you use the **TFFS_STD_FORMAT_PARAMS** macro, the default values will format the entire flash medium for use with TrueFFS. The most common reason for changing **formatParams** is to support a boot region. If you want to create a boot image region that excludes TrueFFS, you need to modify these default values by changing only the first member of the **tffsFormatParams** structure, **bootImageLen**. For details, see *8.7 Creating a Region for Writing a Boot Image*, p.309.

formatFlags Member

The second member of the **tffsDevFormatParams** structure, **formatFlags**, determines the option used to format the drive. There are several possible values for **formatFlags**, which are listed in Table 8-1.

Table 8-1 **Option for formatFlags**

Macro	Value	Meaning
FTL_FORMAT	1	FAT and FTL formatting
FTL_FORMAT_IF_NEEDED	2	FAT formatting, FTL formatting if needed
NO_FTL_FORMAT	0	FAT formatting only

The default macro **TFFS_STD_FORMAT_PARAMS** passes **FTL_FORMAT_IF_NEEDED** as the value for this argument.

8.7 Creating a Region for Writing a Boot Image

Although the translation services of TrueFFS provide many advantages for managing the data associated with a file system, those same services also complicate the use of flash memory as a boot device. The only practical solution to first create a boot image region that excludes TrueFFS, and then write the boot image to that region. This section describes, first, the technical details of the situation, then, how to create the region, and finally, how to write the boot image to it.

8.7.1 Write Protecting Flash

TrueFFS requires that all flash devices that interact with the file system, including boot image regions and NVRAM regions, not be write protected via the MMU. This is because it is essential to the proper working of the product that all commands being issued to the device reach it. Write-protecting the device would impact this behavior, since it would also prevent commands being issued, that are not write-oriented, from reaching the device. For related information, see *rfaWriteProtected*, p.319.

You can, however, reserve a fallow region that is not tampered with by the file system. TrueFFS supports boot code by allowing the user to specify a fallow area when formatting the device. Fallow areas are always reserved at the start of the flash. There have been a few instances where architecture ports have required the fallow area to be at the end of flash. This was accomplished by faking the size of the identification process in the MTD (that is, by telling the MTD that it has less memory than is actually available). The format call is then told that no fallow area is required. TrueFFS does not care how the fallow area is managed, nor is it affected by any faking.

8.7.2 Creating the Boot Image Region

To create the boot image region, format the flash memory so that the TrueFFS segment starts at an offset. This creates a fallow area within flash that is not formatted for TrueFFS. This preserves a boot image region. If you want to update the boot image, you can write a boot image into this fallow area, as described in *8.7.3 Writing the Boot Image to Flash*, p.311.

Formatting at an Offset

To format the flash at an offset, you need to initialize the **tffsFormatParams** structure to values that leave a space on the flash device for a boot image. You do this by specifying a value for the **bootImageLen** member (of the structure) that is at least as large as the boot image. The **bootImageLen** member specifies the offset after which to format the flash medium for use with TrueFFS. For details on **bootImageLen** and other members of the structure, see the comments in the header file *installDir***\target\h\tffs\tffsDrv.h**.

The area below the offset determined by **bootImageLen** is excluded from TrueFFS. This special region is necessary for boot images because the normal translation and wear-leveling services of TrueFFS are incompatible with the needs of the boot program and the boot image it relies upon. When **tffsDevFormat( )** formats flash, it notes the offset, then erases and formats all erase units with starting addresses higher than the offset. The erase unit containing the offset address (and all previous erase units) are left completely untouched. This preserves any data stored before the offset address.

For more information on wear leveling, see *8.13 Flash Memory Functionality*, p.338.

Using a BSP Helper Routine

Some BSPs provide an optional, BSP-specific, helper routine, **sysTffsFormat( )**, which can be called externally to create or preserve the boot image region. This routine first sets up a pointer to a **tffsFormatParams** structure that has been initialized with a value for **bootImageLen** that formats at an offset, creating the boot image region; then it calls **tffsDevFormat( )**.

Several BSPs, among them the ads860 BSP, include a **sysTffsFormat( )** routine that reserves 0.5 MB for the boot image. Following is an example:

```
STATUS sysTffsFormat (void)
    {
    STATUS status;
    tffsDevFormatParams params =
        {
#define HALF_FORMAT
/* lower 0.5MB for bootimage, upper 1.5MB for TFFS */

#ifdef  HALF_FORMAT
    {0x800001, 99, 1, 0x10000l, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#else
    {0x000000l, 99, 1, 0x10000l, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#endif  /* HALF_FORMAT */
```

```
    FTL_FORMAT_IF_NEEDED
    };

/* we assume that the drive number 0 is SIMM */

    status = tffsDevFormat (0, (int)&params);
    return (status);
    }
```

For examples of **sysTffsFormat( )** usage, see the socket drivers in *installDir***/target/src/drv/tffs/sockets**. If your BSP does not provide a **sysTffsFormat( )** routine, then either create a similar routine, or pass the appropriate argument to **tffsDevFormat( )**.

8.7.3 Writing the Boot Image to Flash

If you have created a boot image region, write the boot image to the flash device. To do this you use **tffsBootImagePut( )**, which bypasses TrueFFS (and its translation layer) and writes directly into any location in flash memory. However, because **tffsBootImagePut( )** relies on a call to **tffsRawio( )**, you cannot use this routine once the TrueFFS volume is mounted.

WARNING: Because **tffsBootImagePut( )** lets you write directly to any area of flash, it is possible to accidentally overwrite and corrupt the TrueFFS-managed area of flash. For more information about how to carefully use this utility, see the reference entry for **tffsBootImagePut( )** in the *VxWorks API Reference*.

The **tffsBootImagePut( )** routine is defined in *installDir***/target/src/drv/tffs/tffsConfig.c** as:

```
STATUS tffsBootImagePut
    (
    int    driveNo,                /* TFFS drive number */
    int    offset,                 /* offset in the flash chip/card */
    char * filename                /* binary format of the bootimage */
    )
```

This routine takes as arguments:

- The *driveNo* parameter is the same drive number as the one used as input to the format routine.
- The *offset* parameter is the actual offset from the start of flash at which the image is written (most often specified as zero).
- The *filename* parameter is a pointer to the boot image (bootApp or boot ROM image).

NOTE: For a detailed description of the **bootImagePut( )** routine, see the comments in *installDir***/target/src/drv/tffs/tffsConfig.c**.

8.8 Mounting the Drive

Next, use the **usrTffsConfig( )** routine to mount the VxWorks DOS file system on a TrueFFS flash drive. This routine is defined in *installDir***/target/config/comps/src/usrTffs.c** as:

```
STATUS usrTffsConfig
    (
    int    drive,                /* drive number of TFFS */
    int    removable,            /* 0 for nonremovable flash media */
    char * fileName              /* mount point */
    )
```

This routine takes three arguments:

- The *drive* parameter specifies the drive number of the TFFS flash drive; valid values are 0 through the number of socket interfaces in BSP.
- The *removable* parameter specifies whether the media is removable. Use 0 for non-removable, 1 for removable.
- The *fileName* parameter specifies the mount point, for example, '**/tffs0/**'.

The following example runs **usrTffsConfig( )** to attach a drive to dosFs, and then runs **devs** to list all drivers:

```
% usrTffsConfig 0,0,"/flashDrive0/"

% devs
drv     name
0       /null
1       /tyCo/0
1       /tyCo/1
5       host:
6       /vio
2       /flashDrive0/
```

Internally, **usrTffsConfig( )** calls other routines, passing the parameters you input.

Among these routines is **tffsDevCreate( )**, which creates a TrueFFS block device on top of the socket driver. This routine takes, as input, a number (0 through 4, inclusive) that identifies the socket driver on top of which to construct the TrueFFS block device. The **tffsDevCreate( )** call uses this number as an index into the array of **FLSocket** structures. This number is visible later to dosFs as the driver number.

After the TrueFFS block device is created, **dcacheDevCreate( )** and then **dosFsDevCreate( )** are called. This routine mounts dosFs onto the device. After mounting dosFs, you can read and write from flash memory just as you would from a standard disk drive.

8.9 Running the Shell Commands with Examples

Each of these examples assumes that you have built VxWorks and booted the target.

assabet with a Board-Resident Flash Array and a Boot Image

This example uses **sysTffsFormat( )** to format board-resident flash, preserving the boot image region. It does not update the boot image, so no call is made to **tffsBootImagePut( )**. Then, it mounts the non-removable RFA medium as drive number 0.

At the target shell prompt, enter the following commands:

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
```

ads860 with a Board-Resident Flash Array and a PCMCIA Slot

This example formats RFA and PCMCIA flash for two drives.

The first lines of this example format the board-resident flash by calling the helper routine, **sysTffsFormat( )**, which preserves the boot image region. This example does not update the boot image. It then mounts the drive, numbering it as 0 and passing 0 as the second argument to **usrTffsConfig( )**. Zero is used because RFA is non-removable.

The last lines of the example format PCMCIA flash, passing default format values to **tffsDevFormat( )** for formatting the entire drive. Then, it mounts that drive. Because PCMCIA is removable flash, it passes 1 as the second argument to **usrTffsConfig( )**. (See *8.8 Mounting the Drive*, p.312 for details on the arguments to **usrTffsConfig( )**.)

Insert a flash card in the PCMCIA socket. At the target shell prompt, enter the following commands:

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA1/"
```

mv177 with a Board-Resident Flash Array and No Boot Image Region Created

This example formats board-resident flash using the default parameters to **tffsDevFormat( )**, as described in *8.6 Formatting the Device*, p.306. Then, it mounts the drive, passing 0 as the drive number and indicating that the flash is non-removable.

At the target shell prompt, enter the following commands:

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,0,"/RFA/"
```

x86 with Two PCMCIA Slots Using INCLUDE_PCMCIA

This example formats PCMCIA flash for two drives. Neither format call preserves a boot image region. Then, it mounts the drives, the first is numbered 0, and the second is numbered 1. PCMCIA is a removable medium.

Insert a flash card in each PCMCIA socket. At the target shell prompt, enter the following commands:

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,1,"/PCMCIA1/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA2/"
```

8.10 Writing Socket Drivers

The socket driver is implemented in the file **sysTffs.c**. TrueFFS provides a stub version of the socket driver file for BSPs that do not include one. As a writer of the socket driver, your primary focus is on the following key contents of the socket driver file:

- The **sysTffsInit( )** routine, which is the main routine. This routine calls the socket registration routine.
- The *xxx***Register( )** routine, which is the socket registration routine. This routine is responsible for assigning routines to the member functions of the socket structure.
- The routines assigned by the registration routine.
- The macro values that should reflect your hardware.

In this stub file, all of the required routines are declared. Most of these routines are defined completely, although some use generic or fictional macro values that you may need to modify.

The socket register routine in the stub file is written for RFA (Resident Flash Array) sockets only. There is no stub version of the registration routine for PCMCIA socket drivers. If you are writing a socket driver for RFA, you can use this stub file and follow the steps described in *8.10.1 Porting the Socket Driver Stub File*, p.315. If you are writing a PCMCIA socket driver, see the example in *installDir***/target/src/drv/tffs/sockets/pc386-sysTffs.c** and the general information in *8.10.2 Understanding Socket Driver Functionality*, p.319.

NOTE: Examples of other RFA socket drivers are in *installDir***/target/src/drv/tffs/sockets**.

8.10.1 Porting the Socket Driver Stub File

If you are writing your own socket driver, it is assumed that your BSP does not provide one. When you run the build, a stub version of the socket driver, **sysTffs.c**, is copied from *installDir***/target/config/comps /src** to your BSP directory. Alternatively, you can copy this version manually to your BSP directory before you run a build. In either case, edit only the file copied to the BSP directory; do not modify the original stub file.

This stub version is the starting point for you, to help you port the socket driver to your BSP. As such, it contains incomplete code and does not compile. The modifications you need to make are listed below. They are not extensive and all are noted by **/* TODO */** clauses.

1. Replace "fictional" macro values, such as **FLASH_BASE_ADRS**, with correct values that reflect your hardware. Then, remove the following line:

   ```
   #error "sysTffs: Verify system macros and function before first use"
   ```

2. Add calls to the registration routine for each additional device (beyond one) that your BSP supports. Therefore, if you have only one device, you do not need to do anything for this step. For details, see *Call the Socket Register Routines*, p.316.
3. Review the implementation for the two routines marked **/* TODO */**. You may or may not need to add code for them. For details, see *Implement the Socket Structure Member Functions*, p.316.

⚠ **CAUTION:** Do not edit the original copy of the stub version of **sysTffs.c** in *installDir*/**target/config/comps /src**, since you may need it for future ports.

Call the Socket Register Routines

The main routine in **sysTffs.c** is **sysTffsInit( )**, which is automatically called at boot time. The last lines of this routine call the socket register routines for each device supported by your system. The stub **sysTffs.c** file specifically calls the socket register routine **rfaRegister( )**.

If your BSP supports only one (RFA) flash device, you do not need to edit this section. However, if your BSP supports several flash devices, you are to edit the stub file to add calls for each socket's register routine. The place to do this is indicated by the **/* TODO */** comments in the **sysTffsInit( )** routine.

If you have several socket drivers, you can encapsulate each *xxx***Register( )** call in pre-processor conditional statements, as in the following example:

```
#ifdef  INCLUDE_SOCKET_PCIC0
    (void) pcRegister (0, PC_BASE_ADRS_0);   /* flash card on socket 0 */
#endif  /* INCLUDE_SOCKET_PCIC0 */

#ifdef  INCLUDE_SOCKET_PCIC1
    (void) pcRegister (1, PC_BASE_ADRS_1);   /* flash card on socket 1 */
#endif  /* INCLUDE_SOCKET_PCIC1 */
```

Define the constants in the BSP's **sysTffs.c**. Then, you can use them to selectively control which calls are included in **sysTffsInit( )** at compile time.

Implement the Socket Structure Member Functions

The stub socket driver file also contains the implementation for the **rfaRegister( )** routine, which assigns routines to the member functions of the **FLSocket** structure, **vol**. TrueFFS uses this structure to store the data and function pointers that handle the hardware (socket) interface to the flash device. For the most part, you need not be concerned with the **FLSocket** structure, only with the routines assigned to it. Once these routines are implemented, you never call them directly; they are called automatically by TrueFFS.

All of the routines assigned to the socket structure member functions by the registration routine are defined in the stub socket driver module. However, only the **rfaSocketInit( )** and **rfaSetWindow( )** routines are incomplete. When you are editing the stub file, note the **#error** and **/* TODO */** comments in the code. These indicate where and how you modify the code.

Following is a list of all of the routines assigned by the registration routine, along with a description of how each was implemented in the stub file. The two routines that require your attention are listed with descriptions of how they are to be implemented.

NOTE: More detailed information on the functionality of each routine is provided in *8.10.2 Understanding Socket Driver Functionality*, p.319. However, this information is not necessary for you to port the socket driver.

rfaCardDetected

This routine always returns TRUE in RFA environments, since the device is not removable. Implementation is complete in the stub file.

rfaVccOn

Vcc must be known to be good on exit. It is assumed to be **ON** constantly in RFA environments. This routine is simply a wrapper. While the implementation is complete in the stub file, you may want to add code as described below.

When switching Vcc on, the **VccOn( )** routine must not return until Vcc has stabilized at the proper operating voltage. If necessary, your function should delay execution with an idle loop, or with a call to the **flDelayMsec( )** routine, until the Vcc has stabilized.

rfaVccOff

Vcc is assumed to be **ON** constantly in RFA environments. This routine is simply a wrapper and is complete in the stub file.

rfaVppOn

Vpp must be known to be good on exit and is assumed to be **ON** constantly in RFA environments. This routine is not optional, and must always be implemented.

Therefore, do not delete this routine. While the implementation in the stub file is complete, you may want to add code, as described below.

While When switching Vpp on, the **VppOn( )** function must not return until Vpp has stabilized at the proper voltage. If necessary, your **VppOn( )** function should delay execution with an idle loop or with a call to the **flDelayMsec( )** routine, until the Vpp has stabilized.

rfaVppOff

Vpp is assumed to be **ON** constantly in RFA environments. This routine is complete in the stub file; however, it is not optional, and must always be implemented. Therefore, do not delete this routine.

rfaSocketInit

Contains a **/* TODO */** clause.

This routine is called each time TrueFFS is initialized (the drive is accessed). It is responsible for ensuring that the flash is in a usable state (that is, board-level initialization). If, for any reason, there is something that must be done prior to such an access, this is the routine in which you perform that action. For more information, see *nrfaSocketInit*, p.321.

rfaSetWindow

Contains a **/* TODO */** clause.

This routine uses the **FLASH_BASE_ADRS** and **FLASH_SIZE** values that you set in the stub file. As long as those values are correct, the implementation for this routine in the stub file is complete.

TrueFFS calls this routine to initialize key members of the **window** structure, which is a member of the **FLSocket** structure. For most hardware, the **setWindow** function does the following, which is already implemented in the stub file:

- Sets the **window.baseAddress** to the base address in terms of 4 KB pages.
- Calls **flSetWindowSize( )**, specifying the window size in 4 KB units (**window.baseAddress**). Internally, the call to **flSetWindowSize( )** sets **window.size**, **window.base**, and **window.currentPage** for you.

This routine sets current window hardware attributes: base address, size, speed and bus width. The requested settings are given in the **vol.window** structure. If it is not possible to set the window size requested in **vol.window.size**, the window size should be set to a larger value, if possible. In any case, **vol.window.size** should contain the actual window size (in 4 KB units) on exit.

For more information, see *nrfaSetWindow*, p.321 and *Socket Windowing and Address Mapping*, p.322.

⚠ **CAUTION:** On systems with multiple socket drivers (to handle multiple flash devices), make sure that the window base address is different for each socket. In addition, the window size must be taken into account to verify that the windows do not overlap.

rfaSetMappingContext

TrueFFS calls this routine to set the window mapping register. Because board-resident flash arrays usually map the entire flash in memory, they do not need this function. In the stub file it is a wrapper, thus implementation is complete.

rfaGetAndClearChangeIndicator

Always return FALSE in RFA environments, since the device is not removable. This routine is complete in the stub file.

rfaWriteProtected

This routine always returns FALSE for RFA environments. It is completely implemented in the stub file.

8.10.2 Understanding Socket Driver Functionality

Socket drivers in TrueFFS are modeled after the PCMCIA socket services. As such, they must provide the following:

- services that control power to the socket (be it PCMCIA, RFA, or any other type)
- criteria for setting up the memory windowing environment
- support for card change detection
- a socket initialization routine

This section describes details about socket registration, socket member functions, and the windowing and address mapping set by those functions. This information is not necessary to port the stub RFA file; however, it may be useful for writers of PCMCIA socket drivers.

Socket Registration

The first task the registration routine performs is to assign drive numbers to the socket structures. This is fully implemented in the stub file. You only need to be aware of the drive number when formatting the drives (*8.6.1 Specifying the Drive Number*, p.307).

The drive numbers are index numbers into a pre-allocated array of **FLSocket** structures. The registration sequence dictates the drive number associated with a drive, as indicated in the first line of code from the **rfaRegister( )** routine:

```
    FLSocket vol = flSockeOf (noOfDrives);
```

Here, **noOfDrives** is the running count of drives attached to the system. The function **flSocketOf( )** returns a pointer to socket structure, which is used as the volume description and is incremented by each socket registration routine called by the system. Thus, the TrueFFS core in the socket structures are allocated each of the (up to) 5 drives supported for the system.[2] When TrueFFS invokes the routines that you implement to handle its hardware interface needs, it uses the drive number as an index into the array to access the socket hardware for a particular flash device.

Socket Member Functions

- **rfaCardDetected** This routine reports whether there is a flash memory card in the PCMCIA slot associated with this device. For non-removable media, this routine should always return TRUE. Internally, TrueFFS for Tornado calls this function every 100 milliseconds to check that flash media is still there. If this function returns FALSE, TrueFFS sets **cardChanged** to TRUE.

- **rfaVccOn** TrueFFS can call this routine to turn on *Vcc*, which is the operating voltage. For the flash memory hardware, Vcc is usually either 5 or 3.3 Volts. When the media is idle, TrueFFS conserves power by turning Vcc off at the completion of an operation. Prior to making a call that accesses flash memory, TrueFFS uses this function to turn the power back on again.

  However, when socket polling is active, a delayed Vcc-off mechanism is used, in which Vcc is turned off only after at least one interval has passed. If several flash-accessing operations are executed in rapid sequence, Vcc remains on during the sequence, and is turned off only when TrueFFS goes into a relatively idle state.

[2] TrueFFS only supports a maximum of 5 drives.

- **rfaVccOff** TrueFFS can call this routine to turn off the operating voltage for the flash memory hardware. When the media is idle, TrueFFS conserves power by turning Vcc off. However, when socket polling is active, Vcc is turned off only after a delay. Thus, if several flags accessing operations are executed in rapid sequence, Vcc is left on during the sequence. Vcc is turned off only when TrueFFS goes into a relatively idle state. Vcc is assumed to be **ON** constantly in RFA environments.

- **rfaVppOn** This routine is not optional, and must always be implemented. TrueFFS calls this routine to apply *Vpp*, which is the programming voltage. Vpp is usually 12 Volts to the flash chip. Because not all flash chips require this voltage, the member is included only if **SOCKET_12_VOLTS** is defined.

  Vpp must be known to be good on exit and is assumed to be **ON** constantly in RFA environments.

NOTE: The macro **SOCKET_12_VOLTS** is only alterable by users that have source to the TrueFFS core.

- **rfaVppOff** TrueFFS calls this routine to turn off a programming voltage (Vpp, usually 12 Volts) to the flash chip. Because not all flash chips require this voltage, the member is included only if **SOCKET_12_VOLTS** is defined. This routine is not optional, and must always be implemented. Vpp is assumed to be **ON** constantly in RFA environments.

- **rfaSocketInit** TrueFFS calls this function before it tries to access the socket. TrueFFS uses this function to handle any initialization that is necessary before accessing the socket, especially if that initialization was not possible at socket registration time. For example, if you did no hardware detection at socket registration time, or if the flash memory medium is removable, this function should detect the flash memory medium and respond appropriately, including setting **cardDetected** to FALSE if it is missing.

- **rfaSetWindow** TrueFFS uses **window.base** to store the base address of the memory window on the flash memory, and **window.size** to store the size of the memory window. TrueFFS assumes that it has exclusive access to the window. That is, after it sets one of these window characteristics, it does not expect your application to directly change any of them, and could crash if you do. An exception to this is the mapping register. Because TrueFFS always reestablishes this register when it accesses flash memory, your application may map the window for purposes other than TrueFFS. However, do not do this from an interrupt routine.

- **rfaSetMappingContext** TrueFFS calls this routine to set the window mapping register. This routine performs the sliding action by setting the mapping register to an appropriate value. Therefore, this routine is meaningful only in environments such as PCMCIA, that use the sliding window mechanism to view flash memory. Flash cards in the PCMCIA slot use this function to access/set a mapping register that moves the effective flash address into the host's memory window. The mapping process takes a "card address", an offset in flash, and produces real address from it. It also wraps the address around to the start of flash if the offset exceeds flash length. The latter is the only reason why the flash size is a required entity in the socket driver. On entry to **setMappingContext**, **vol.window.currentPage** is the page already mapped into the window (meaning that it was mapped in by the last call to **setMappingContext**).

- **rfaGetAndClearChangeIndicator** This routine reads the hardware card-change indication and clears it. It serves as a basis for detecting media-change events. If you have no such hardware capability, return FALSE for this routine (set this function pointer to **NULL)**.

- **rfaWriteProtected** TrueFFS can call this routine to get the current state of the media's write-protect switch (if available). This routine returns the write-protect state of the media, if available, and always returns FALSE for RFA environments. For more information, see *8.7.1 Write Protecting Flash*, p.309.

Socket Windowing and Address Mapping

The **FLSocket** structure (defined in *installDir***/target/h/tffs/flsocket.h**) contains an internal **window** state structure. If you are porting the socket driver, the following background information about this **window** structure may be useful when implementing the *xxx***SetWindow( )** and *xxx***SetMappingContext( )** routines.

The concept of windowing derives from the PCMCIA world, which formulated the idea of a Host Bus Adapter. The host could allow one of the following situations to exist:

- The PCMCIA bus could be entirely visible in the host's address range.
- Only a segment of the PCMCIA address range could be visible in the host's address space.
- Only a segment of the host's address space could be visible to the PCMCIA.

To support these concepts, PCMCIA specified the use of a "window base register" that may be altered to adjust the view from the window. In typical RFA scenarios, where the device logic is NOR, the window size is that of the amount of flash on the board. In the PCMCIA situation, the window size is implementation-specific. The book *PCMCIA Systems Architecture* by Don Anderson provides an good explanation of this concept, with illustrations.

8.11 Using the MTD-Supported Flash Devices

This section lists the flash devices that are supported by the MTDs that are provided with the product.

8.11.1 Supporting the Common Flash Interface (CFI)

TrueFFS supports devices that use the *Scalable Command Set* (SCS) from Intel, and devices that use the AMD command set. Both of these command sets conform to the *Common Flash Interface* (CFI). System that requires support for both command sets are rare. Therefore, to facilitate code readability, support for each command set is provided in a separate MTD. To support both command sets, simply configure your system to include both MTDs (see *8.5.4 Including the MTD Component*, p.303). The command sets are described below.

- **Intel/Sharp Command Set .** This is the CFI specification listing for the SCS command set. The driver file for this MTD is *installDir***/target/src/drv/tffs/cfiscs.c**. The support for the Intel/Sharp command set was largely derived from the *Application Note 646*, available at the Intel web site.

- **AMD/Fujitsu Command Set .** This is the *Embedded Program Algorithm* and flexible sector architecture listing for the SCS command set. The driver file is *installDir***/target/src/drv/tffs/cfiamd.c**. Details about support for this MTD are described in *AMD/Fujitsu CFI Flash Support*, p.325.

Common Functionality

Both drivers support 8 and 16 bit devices, and 8- and 16-bit wide interleaves. Configuration macros (described in the code) are used to control these and other configuration issues, and must be defined specifically for your system. If modifications are made to the code, it must be rebuilt. Noteworthy are the following macros:

INTERLEAVED_MODE_REQUIRES_32BIT_WRITES
Must be defined for systems that have 16-bit interleaves and require support for the "Write to Buffer" command.

SAVE_NVRAM_REGION
Excludes the last erase block on each flash device in the system that is used by TrueFFS; this is so that the region can be used for Non-Volatile Storage of boot parameters.

CFI_DEBUG
Makes the driver verbose by using the I/O routine defined by **DEBUG_PRINT**.

BUFFER_WRITE_BROKEN
Introduced to support systems that registered a buffer size greater than 1, yet could not support writing more than a byte or word at a time. When defined, it forces the buffer size to 1.

DEBUG_PRINT
If defined, makes the driver verbose by using its value.

NOTE: These macros are only configurable by defining them in the source file, not through the IDE project tool.

CFI/SCS Flash Support

The MTD defined in **cfiscs.c** supports flash components that follow the CFI/SCS specification. CFI stands for Common Flash Interface. SCS stands for Scalable Command Set. CFI is a standard method for querying flash components for their characteristics. SCS is a second layer built on the CFI specification. This lets a single MTD handle all CFI/SCS flash technology in a common manner.

The joint CFI/SCS specification is currently adopted by Intel Corporation and Sharp Corporation for all new flash components starting in 1997.

The CFI document can be downloaded from:

http://www.intel.com/design/flcomp/applnots/292204.htm

or can be found by searching for CFI at:

http://www.intel.com/design

Define **INCLUDE_MTD_CFISCS** in your BSP's **sysTffs.c** file to include this MTD in TrueFFS for Tornado.

In some of the more recent target boards we have observed that non-volatile RAM circuitry does not exist, and that the BSP developers have opted to use the high end of flash for this purpose. The last erase block of each flash part is used to make up this region. The CFISCS MTD supports this concept by providing the compiler constant **SAVE_NVRAM_REGION**. If this is defined, the driver reduces the device's size by the erase block size times the number of devices; this results in the NVRAM region being preserved and never over-written. The ARM BSPs, in particular, use flash for NVRAM and for the boot image.

AMD/Fujitsu CFI Flash Support

In AMD and Fujitsu devices, the flexible sector architecture, also called *boot block* devices, is only supported when erasing blocks. However, the TrueFFS core and translation layers have no knowledge of the subdivision within the boot block because the MTD presents this division transparently. According to the data sheet for the 29LV160 device, it is comprised of 35 sectors. However, the 4 boot block sectors appear to the core and translation layer as yet another, single (64 KB) sector. Thus, the TrueFFS core detects only 32 sectors. Consequently, the code that supports boot images also has no knowledge of the boot block, and cannot provide direct support for it.

The AMD and Fujitsu devices include the concept of **Top** and **Bottom** boot devices. However, the CFI interrogation process does not provide a facility for distinguishing between the two. Thus, in order to determine the boot block type the driver embeds the JEDEC device IDs in it. This limits the number of supported devices to the ones that are registered in it, requiring verification that the device in use is listed in the registry.

8.11.2 Supporting Other MTDs

If you are not using a CFI-compliant MTD, the following MTDs are also provided.

Intel 28F016 Flash Support

The MTD defined in **i28f016.c** supports Intel 28F016SA and Intel 28F008SV flash components. Any flash array or card based on these chips is recognized and supported by this MTD. This MTD also supports interleaving factors of 2 and 4 for BYTE-mode 28F016 component access.

For WORD-mode component access, only non-interleaved (interleave 1) mode is supported. The list of supported flash media includes the following:

- Intel Series-2+ PC Cards
- M-Systems Series-2+ PC Cards

Define **INCLUDE_MTD_I28F016** in your BSP's **sysTffs.c** file to include this MTD in TrueFFS for Tornado.

Intel 28F008 Flash Support

The MTD defined in **I28F008.c** supports the Intel 28F008SA, Intel 28F008SC, and Intel 28F016SA/SV (in 8-mbit compatibility mode) flash components. Any flash array or card based on these chips is recognized and supported by this MTD. However, the WORD-mode of 28F016SA/SV is not supported (BYTE-mode only). This MTD also supports all interleaving factors (1, 2, 4, ...). Interleaving of more than 4 is recognized, although the MTD does not access more than 4 flash parts simultaneously. The list of supported flash media includes the following:

- M-Systems D-Series PC Cards
- M-Systems S-Series PC Cards
- Intel Series-2 (8-mbit family) PC Cards
- Intel Series-2+ (16-mbit family) PC Cards
- Intel Value Series 100 PC Cards
- Intel Miniature cards
- M-Systems PC-FD, PC-104-FD, Tiny-FD flash disks

Define **INCLUDE_MTD_I28F008** in your BSP's **sysTffs.c** file to include this MTD in TrueFFS for Tornado.

AMD/Fujitsu Flash Support

The MTD defined in **amdmtd.c** (8-bit) supports AMD flash components of the AMD Series-C and Series-D flash technology family, as well as the equivalent Fujitsu flash components. The flash types supported are:

- Am29F040 (JEDEC IDs 01a4h, 04a4h)
- Am29F080 (JEDEC IDs 01d5h, 04d5h)
- Am29LV080 (JEDEC IDs 0138h, 0438h)
- Am29LV008 (JEDEC IDs 0137h, 0437h)
- Am29F016 (JEDEC IDs 01adh, 04adh)
- Am29F016C (JEDEC IDs 013dh, 043dh)

Any flash array or card based on these chips is recognized and supported by this MTD. The MTD supports interleaving factors of 1, 2, and 4. The list of supported flash media includes the following:

- AMD and Fujitsu Series-C PC cards
- AMD and Fujitsu Series-D PC cards
- AMD and Fujitsu miniature cards

Define **INCLUDE_MTD_AMD** in your BSP's **sysTffs.c** file to include the 8-bit MTD in TrueFFS for Tornado.

8.11.3 Obtaining Disk On Chip Support

The previous demand for NAND devices has been in one of two forms: SSFDC/ Smart Media devices and Disk On Chip from M-Systems. Each of these is supported by a separate translation layer. To provide M-Systems the capability of adding Disk On Chip specific optimizations within the product that do not effect other supported devices, support for M-Systems devices must now be obtained directly from M-Systems and is no longer distributed with the Tornado product. This version of Tornado only supports NAND devices that conform to the SSFDC specification. *8.5.5 Including the Translation Layer*, p.304.

8.12 Writing MTD Components

An MTD is a software module that provides TrueFFS with data, and with pointers to the routines that it uses to program the flash memory. All MTDs must provide the following three routines: a write routine, an erase routine, and an identification routine. The MTD module uses an identification routine to evaluate whether the type of the flash device is appropriate for the MTD. If you are writing your own MTD, you need to define it as a component and register the identification routine.

For source code examples of MTDs, see the *installDir*/**target/src/drv/tffs** directory.

8.12.1 Writing the MTD Identification Routine

TrueFFS provides a flash structure in which information about each flash part is maintained. The identification process is responsible for setting up the flash structure correctly.

NOTE: Many of the MTDs previously developed by M-Systems or Wind River are provided in source form as examples of how one might write an MTD (in *installDir***/target/src/drv/tffs**). This section provides additional information about writing them.

In the process of creating a logical block device for a flash memory array, TrueFFS tries to match an MTD to the flash device. To do this, TrueFFS calls the identification routine from each MTD until one reports a match. The first reported match is the one taken. If no MTD reports a match, TrueFFS falls back on a default read-only MTD that reads from the flash device by copying from the socket window.

The MTD identification routine is guaranteed to be called prior to any other routine in the MTD. An MTD identification routine is of the following format:

```
FLStatus xxxIdentify(FLFlash vol)
```

Within an MTD identify routine, you must probe the device to determine its type. How you do this depends on the hardware. If the type is not appropriate to this MTD, return failure. Otherwise, set the members of the **FLFlash** structure listed below.

The identification routine for every MTD must be registered in the **mtdTable[ ]** defined in *installDir***/target/src/drv/tffs/tffsConfig.c**. Each time a volume is mounted, the list of identification routines is traversed to find the MTD suitable for the volume. This provides better service for hot-swap devices; no assumption is made about a previously identified device being the only device that will work for a given volume.

Device identification can be done in a variety of ways. If your device conforms to Joint Electronic Device Engineering Council (JEDEC) or Common Flash Interface (CFI) standards, you can use their identification process. You might want your MTD to identify many versions of the device, or just simply one.

Initializing the FLFLash Structure Members

At the end of the identification process, the ID routine needs to set all data elements in the **FlFlash** structure, except the **socket** member. The **socket** member is set by functions internal to TrueFFS. The **FLFlash** structure is defined in *installDir*/**h/tffs/flflash.h**. Members of this structure are described as follows:

type

The JEDEC ID for the flash memory hardware. This member is set by the MTD's identification routine.

erasableBlockSize

The size, in bytes, of an erase block for the attached flash memory hardware. This value takes interleaving into account. Thus, when setting this value in an MTD, the code is often of the following form:

```
vol.erasableBlockSize = aValue * vol.interleaving;
```

Where *aValue* is the erasable block size of a flash chip that is not interleaved with another.

chipSize

The size (storage capacity), in bytes, of one of the flash memory chips used to construct the flash memory array. This value is set by the MTD, using your **flFitInSocketWindow( )** global routine.

noOfChips

The number of flash memory chips used to construct the flash memory array.

interleaving

The interleaving factor of the flash memory array. This is the number of devices that span the data bus. For example, on a 32-bit bus we can have four 8-bit devices or two 16-bit devices.

flags

Bits 0-7 are reserved for the use of TrueFFS (it uses these flags to track things such as the volume mount state). Bits 8-15 are reserved for the use of the MTDs.

mtdVars

This field, if used by the MTD, is initialized by the MTD identification routine to point to a private storage area. These are instance-specific. For example, suppose you have an Intel RFA based on the I28F016 flash part; suppose you also have a PCMCIA socket into which you decide to plug a card that has the same flash part. The same MTD is used for both devices, and the **mtdVars** are used for the variables that are instance-specific, so that an MTD may be used more than once in a system.

socket

This member is a pointer to the **FLSocket** structure for your hardware device. This structure contains data and pointers to the socket layer functions that TrueFFS needs to manage the board interface for the flash memory hardware. The functions referenced in this structure are installed when you register your socket driver (see *8.10 Writing Socket Drivers*, p.314). Further, because TrueFFS uses these socket driver functions to access the flash memory hardware, you must register your socket driver *before* you try to run the MTD identify routine that initializes the bulk of this structure.

map

A pointer to the flash memory map function, the function that maps flash into an area of memory. Internally, TrueFFS initializes this member to point to a default map function appropriate for all NOR (linear) flash memory types. This default routine maps flash memory through simple socket mapping. Flash should replace this pointer to the default routine with a reference to a routine that uses map-through-copy emulation.

read

A pointer to the flash memory read function. On entry to the MTD identification routine, this member has already been initialized to point to a default read function that is appropriate for all NOR (linear) flash memory types. This routine reads from flash memory by copying from a mapped window. If this is appropriate for your flash device, leave **read** unchanged. Otherwise, the MTD identify routine should update this member to point to a more appropriate function.

write

A pointer to the flash memory write function. Because of the dangers associated with an inappropriate write function, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member.

erase

A pointer to the flash memory erase function. Because of the dangers associated with an inappropriate erase function, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member.

setPowerOnCallback

A pointer to the function TrueFFS should execute after the flash hardware device powers up. TrueFFS calls this routine when it tries to mount a flash device. Do not confuse this member of **FLFlash** with the **powerOnCallback** member of the **FLSocket** structure. For many flash memory devices, no such function is necessary. However, this member is used by the MTD defined in *installDir***/target/src/drv/tffs/nfdc2048.c**.

Return Value

The identification routine must return **flOK** or an appropriate error code defined in **flbase.h**. The stub provided is:

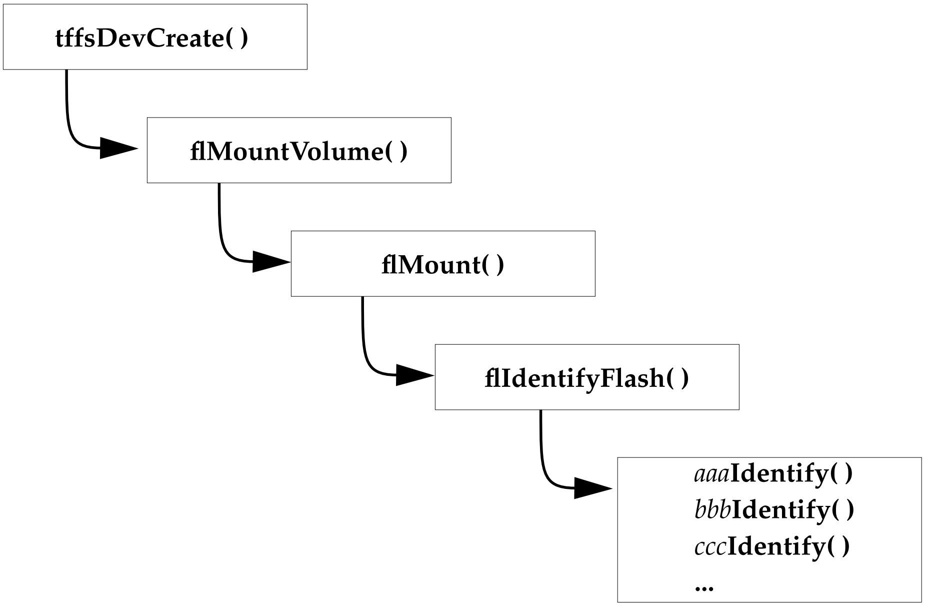
```
FLStatus myMTDIdentification
    (
    FLFlash vol
    )
    {
    /* Do what is needed for identification */

    /* If identification fails return appropriate error */

    return flOK;
    }
```

After setting the members listed above, this function should return **flOK**.

Call Sequence

Upon success, the identification routine updates the **FLFlash** structure, which also completes the initialization of the **FLSocket** structure referenced within this **FLFlash** structure.

Figure 8-3 **Identifying an MTD for the Flash Technology**



8.12.2 Writing the MTD Map Function

MTDs need to provide a map function only when a RAM buffer is required for windowing. No MTDs are provided for devices of this kind in this release. If the device you are using requires such support, you need to add a map function to your MTD and assign a pointer to it in **FLFlash.map**. The function takes three arguments, a pointer to the volume structure, a "card address", and a length field, and returns a void pointer.

```
static void FAR0 * Map
     (FLFlash vol,
      CardAddress address,
      int length
     )
```

```
{
/* implement function */
}
```

8.12.3 Writing the MTD Read, Write, and Erase Functions

Typically, your read, write, and erase functions should be as generic as possible. This means that they should:

- Read, write, or erase only a character, a word, or a long word at a time.
- Be able to handle an unaligned read or write.
- Be able to handle a read, write, or erase that crosses chip boundaries.

When writing these functions, you probably want to use the MTD helper functions **flNeedVpp( )**, **flDontNeedVpp( )**, and **flWriteProtected( )**. The interfaces for these routines are as follows:

```
FLStatus  flNeedVpp(FLSocket vol)
void  flDontNeedVpp(FLSocket vol)
FLBoolean flWriteProtected(FLSocket vol)
```

Use **flNeedVpp( )** if you need to turn on the Vpp (programming voltage) for the chip. Internally, **flNeedVpp( )** bumps a counter, **FLSocket.VppUsers**, and then calls the function referenced in **FLSocket.VppOn**. After calling **flNeedVpp( )**, check its return status to verify that it succeeded in turning on Vpp.

When done with the write or erase that required Vpp, call **flDontNeedVpp( )** to decrement the **FLSocket.VppUsers** counter. This **FLSocket.VppUsers** counter is part of a delayed-off system. While the chip is busy, TrueFFS keeps the chip continuously powered. When the chip is idle, TrueFFS turns off the voltage to conserve power. [3]

Use **flWriteProtected( )** to test that the flash device is not write protected. The MTD write and erase routines must not do any flash programming before checking that writing to the card is allowed. The boolean function **flWriteProtected( )** returns TRUE if the card is write-protected and FALSE otherwise.

Read Routine

If the flash device can be mapped directly into flash memory, it is generally a simple matter to read from it. TrueFFS supplies a default function that performs a

[3] An MTD does not need to touch Vcc. TrueFFS turns Vcc on before calling an MTD function.

remap, and simple memory copy, to retrieve the data from the specified area. However, if the mapping is done through a buffer, you must provide your own read routine.

Write Routine

The write routine must write a given block at a specified address in flash. Its arguments are a pointer to the flash device, the address in flash to write to, a pointer to the buffer that must be written, and the buffer length. The last parameter is boolean, and if set to TRUE implies that the destination has not been erased prior to the write request. The routine is declared as **static** since it is only called from the volume descriptor. The stub provided is:

```
static FLStatus myMTDWrite
    (
    FLFlash vol,
    CardAddress address,
    const void FAR1 *buffer,
    int length,
    FLBoolean overwrite
    )
    {
    /* Write routine */
    return flOK;
    }
```

The write routine must do the following:

- Check to see if the device is write protected.
- Turn on Vpp by calling **flNeedVpp( )**.
- Always "map" the "card address" provided to a **flashPtr** before you write.

When implementing the write routine, iterate through the buffer in a way that is appropriate for your environment. If writes are permitted only on word or double word boundaries, check to see whether the buffer address and the card address are so aligned. Return an error if they are not.

The correct algorithms usually follow a sequence in which you:

- Issue a "write setup" command at the card address.
- Copy the data to that address.
- Loop on the status register until either the status turns **OK** or you time out.

Device data sheets usually provide flow charts for this type of algorithm. AMD devices require an unlock sequence to be performed as well.

The write routine is responsible for verifying that what was written matches the content of the buffer from which you are writing. The file **flsystem.h** has prototypes of compare functions that can be used for this purpose.

Erase Routine

The erase routine must erase one or more contiguous blocks of a specified size. This routine is given a flash volume pointer, the block number of the first erasable block and the number of erasable blocks. The stub provided is:

```
Static FLStatus myMTDErase
    (
    FLFlash vol,
    int     firstBlock,
    int     numOfBlocks
    )
    {
    volatile UINT32 * flashPtr;
    int                     iBlock;

    if (flWriteProtected(vol.socket))
               return flWriteProtected;
    for (iBlock = firstBlock; iBlock < iBlock + numOfBlocks; Iblock++)
        {
        flashPtr = vol.map (&vol, iBlock * vol.erasableBlockSize, 0);

        /* Perform erase operation here */

        /* Verify if erase succeeded */

        /* return flWriteFault if failed*/
        }
    return flOK;
    }
```

As input, the erase can expect a block number. Use the value of the **erasableBlockSize** member of the **FLFlash** structure to translate this block number to the offset within the flash array.

8.12.4 Defining Your MTD as a Component

Once you have completed the MTD, you need to add it as a component to your system project. By convention, MTD components are named **INCLUDE_MTD_***someName*; for example, **INCLUDE_MTD_USR**. You can include the MTD component either through the project facility or, for a command-line configuration and build, by defining it in the socket driver file, **sysTffs.c**.

Adding Your MTD to the Project Facility

In order to have the MTD recognized by the project facility, a component description of the MTD is required. To add your own MTD component to your system by using the project facility, edit the *installDir***\target\config\comps\vxworks\00tffs.cdf** file to include it. MTD components are defined in that file using the following format:

```
Component INCLUDE_MTD_type {
    NAME            name
    SYNOPSIS        type devices
    MODULES         filename.o
    HDR_FILES       tffs/flflash.h tffs/backdrnd.h
    REQUIRES        INCLUDE_TFFS \
                    INCLUDE_TL_type
}
```

Once you define your MTD component in the **00tffs.cdf** file, it appears in the project facility the next time you run Tornado.

Defining the MTD in the Socket Driver File

For a command-line configuration and build, you can include the MTD component simply by defining it in the socket driver file, **sysTffs.c**, as follows:

```
#define INCLUDE_MTD_USR
```

Add your MTD definition to the list of those defined between the conditional clause, as described in *nConditional Compilation*, p.305. Then, define the correct translation layer for your MTD. If both translation layers are defined in the socket driver file, undefine the one you are not using. If both are undefined, define the correct one. For other examples, see the *type*-**sysTffs.c** files in *installDir***\target\src\drv\tffs\sockets**.

CAUTION: Be sure that you have the correct **sysTffs.c** file before changing the defines. For more information, see *8.10.1 Porting the Socket Driver Stub File*, p.315.

8.12.5 Registering the Identification Routine

The identification routine for every MTD must be registered in the **mtdTable[ ]**. Each time a volume is mounted, TrueFFS searches this list to find an MTD suitable for the volume (flash device). For each component that has been defined for your system, TrueFFS executes the identification routine referenced in **mtdTable[ ]**, until it finds a match to the flash device. The current **mtdTable[ ]** as defined in *installDir***/target/src/drv/tffs/tffsConfig.c** is:

```
MTDidentifyRoutine mtdTable[] =         /* MTD tables */
    {
#ifdef  INCLUDE_MTD_I28F016
    i28f016Identify,
#endif                    /* INCLUDE_MTD_I28F016 */

#ifdef  INCLUDE_MTD_I28F008
    i28f008Identify,
#endif                    /* INCLUDE_MTD_I28F008 */

#ifdef  INCLUDE_MTD_AMD
    amdMTDIdentify,
#endif                    /* INCLUDE_MTD_AMD */

#ifdef  INCLUDE_MTD_CDSN
    cdsnIdentify,
#endif                    /* INCLUDE_MTD_CDSN */

#ifdef  INCLUDE_MTD_DOC2
    doc2Identify,
#endif                    /* INCLUDE_MTD_DOC2 */

#ifdef  INCLUDE_MTD_CFISCS
    cfiscsIdentify,
#endif                    /* INCLUDE_MTD_CFISCS */
    };
```

If you write a new MTD, list its identification routine in **mtdTable[ ]**. For example:

```
#ifdef INCLUDE_MTD_USR
    usrMTDIdenitfy,
#endif                    /* INCLUDE_MTD_USR */
```

It is recommended that you surround the component name with conditional include statements, as shown above. The symbolic constants that control these conditional includes are defined in the BSP **config.h** file. Using these constants, your end users can conditionally include specific MTDs.

When you add your MTDs identification routine to this table, you should also add a new constant to the BSP's **config.h**.

8.13 Flash Memory Functionality

This section discusses flash memory functionality, and the ways in which it protects data integrity, extends the lifetime of the medium, and supports fault recovery.

8.13.1 Block Allocation and Data Clusters

As is required of a block device driver, TrueFFS maps flash memory into an apparently contiguous array of storage blocks, upon which a file system can read and write data. These blocks are numbered from zero to one less than the total number of blocks. Currently, the only supported file system is dosFs (see *5.2 MS-DOS-Compatible File System: dosFs*, p. 194).

Block Allocation Algorithm

To promote more efficient data retrieval, TrueFFS uses a flexible allocation strategy, which clusters related data into a contiguous area in a single erase unit. These clusters might be, for example, the blocks that comprise the sectors of a file. TrueFFS follows a prioritized procedure for attempting to cluster the related data. In this order:

(1) First, it tries to maintain a pool of physically consecutive free blocks that are resident in the same erase unit.
(2) If that fails, it then tries to assure that all the blocks in the pool reside in the same erase unit.
(3) If that fails, it finally tries to allocate a pool of blocks in the erase unit that has the most space available.

Benefits of Clustering

Clustering related data in this manner has several benefits, listed and described below.

- **Allows Faster Retrieval Times.** For situations that require TrueFFS to access flash through a small memory window, clustering related data minimizes the number of calls needed to map physical blocks into the window. This allow faster retrieval times for files accessed sequentially.

- **Minimizes Fragmentation.** Clustering related data cuts down on fragmentation because deleting a file tends to free up complete blocks that can be easily reclaimed.
- **Speeds Garbage Collection.** Minimizing fragmentation means that garbage collection is faster.
- **Localizes Static File Blocks.** Localizing blocks that belong to static files significantly facilitates transferring these blocks when the wear-leveling algorithm decides to move static areas.

8.13.2 Read and Write Operations

One of the characteristics of flash memory that differs considerably from the more common magnetic-medium mechanical disks is the way in which it writes new data to the medium. When using traditional magnetic storage media, writing new data to a previously written storage area simply overwrites the existing data, essentially obliterating it; whereas flash does not. This section describes how flash reads from, and writes to, memory.

Reading from Blocks

Reading the data from a block is straightforward. The file system requests the contents of a particular block. In response, TrueFFS translates the block number into flash memory coordinates, retrieves the data at those coordinates, and returns the data to the file system.

Writing to Previously Unwritten Blocks

Writing data to a block is straightforward, if the target block is previously unwritten. TrueFFS translates the block number into flash memory coordinates and writes to the location. However, if the write request seeks to modify the contents of a previously written block, the situation is more complex.

If any write operation fails, TrueFFS attempts a second write. For more information, see *Recovering During a Write Operation*, p.343.

NOTE: Storing data in flash memory requires the use of a manufacturer-supplied programming algorithm, which is defined in the MTD. Consequently, writing to flash is often referred to as programming flash.

Writing to Previously Written Blocks

If the write request is to an area of flash that already contains data, TrueFFS finds a different, writable area of flash instead—one that is already erased and ready to receive data. TrueFFS then writes the new data to that free area. After the data is safely written, TrueFFS updates its block-to-flash mapping structures, so that the block now maps to the area of flash that contains the modified data. This mapping information is protected, even during a fault. For more information on fault recovery and mapping information, see *Recovering Mapping Information*, p.344.

8.13.3 Erase Cycles and Garbage Collection

The write operation is intrinsically linked to the erase operation, since data cannot be "over-written." Data must be erased, and then those erased units must be reclaimed before they are made available to a write operation. This section describes that process, as well as the consequences of over-erasing sections of flash.

Erasing Units

Once data is written to an area of flash memory, modifying blocks leaves behind block-sized regions of flash memory that no longer contain valid data. These regions are also unwritable until erased. However, the erase cycle does not operate on individual bytes or even blocks. Erasure is limited to much larger regions called *erase units*. The size of these erase units depends on the specific flash technology, but a typical size is 64 KB.

Reclaiming Erased Blocks

To reclaim (erase) blocks that no longer contain valid data, TrueFFS uses a mechanism called *garbage collection*. This mechanism copies all the valid data blocks, from a source erase unit, into another erase unit known as a *transfer unit*. TrueFFS then updates the block-to-flash map and afterward erases the old (erase) unit. The virtual block presented to the outside world still appears to contain the same data even though that data now resides in a different part of flash.

For details on the algorithm used to trigger garbage collection, see *Garbage Collection*, p.342. For information about garbage collection and fault recovery, see *Recovering During Garbage Collection*, p.344.

Over-Programming

As a region of flash is constantly erased and rewritten, it enters an *over-programmed* state, in which it responds only very slowly to write requests. With rest, this condition eventually fixes itself, but the life of the flash memory is shortened. Eventually the flash begins to suffer from sporadic erase failures, which become more and more frequent until the medium is no longer erasable and, thus, no longer writable.[4] Consequently, flash limits how often you can erase and rewrite the same area. This number, known as the *cycling limit*, depends on the specific flash technology, but it ranges from a hundred thousand to a million times per block.[5]

8.13.4 Optimization Methods

As mentioned, flash memory is not an infinitely reusable storage medium. The number of erase cycles per erase unit of flash memory is large but limited. Eventually, flash degrades to a read-only state. To delay this as long as possible, TrueFFS uses a garbage collection algorithm that works in conjunction with a technique called wear leveling.

Wear Leveling

One way to alleviate over-programming is to balance usage over the entire medium, so that the flash is evenly worn and no one part is over-used. This technique is known as *wear leveling* and it can extend the lifetime of the flash significantly. To implement wear leveling, TrueFFS uses a block-to-flash translation system that is based on a dynamically maintained map. This map is adjusted as blocks are modified, moved, or garbage collected.

Static-File Locking

By remapping modified blocks to new flash memory coordinates, a certain degree of wear leveling occurs. However, some of the data stored in flash may be essentially static, which means that if wear leveling only occurs during modifications, the areas of flash that store static data are not cycled at all. This situation, known as *static file locking*, exacerbates the wear on other areas of flash,

[4] The data already resident is still readable.
[5] This number is merely statistical in nature and should not be taken as an exact figure.

which must be recycled more frequently as a consequence. If not countered, this situation can significantly lowers the medium's life-expectancy.

TrueFFS overcomes static file locking by forcing transfers of static areas. Because the block-to-flash map is dynamic, TrueFFS can manage these wear-leveling transfers in a way that is invisible to the file system.

Algorithm

Implementing absolute wear leveling can have a negative impact on performance because of all the required data moves. To avoid this performance hit, TrueFFS implements a wear-leveling algorithm that is not quite absolute. This algorithm provides an approximately equal number of erase cycles per unit. In the long run, you can expect the same number of erase cycles per erase unit without a degradation of performance. Given the large number of allowed erase cycles, this less than absolute approach to wear leveling is good enough.

Dead Locks

Finally, the TrueFFS wear-leveling algorithm is further enhanced to break a failure mode known as *dead locks*. Some simple wear-leveling algorithms have been shown to "flip-flop" transfers between only two or more units for very long periods, neglecting all other units. The wear-leveling mechanism used by TrueFFS makes sure that such loops are curbed. For optimal wear-leveling performance, TrueFFS requires at least 12 erase units.

Garbage Collection

Garbage collection, described in *Reclaiming Erased Blocks*, p.340, is used by the erase cycle to reclaim erased blocks. However, if garbage collection is done too frequently, it defeats the wear-leveling algorithm and degrades the overall performance of the flash disk. Thus, garbage collection uses an algorithm that relies on the block allocation algorithm (*Block Allocation Algorithm*, p.338), and is triggered only as needed.

The block allocation algorithm maintains a pool of free consecutive blocks that are resident in the same erase unit. When this pool becomes too small, the block allocation algorithm launches the garbage collection algorithm, which then finds and reclaims an erase unit that best matches the following criteria:

- the largest number of garbage blocks
- the least number of erase cycles
- the most static areas

In addition to these measurable criteria, the garbage collection algorithm also factors in a random selection process. This helps guarantee that the reclamation process covers the entire medium evenly and is not biased due to the way applications use data.

8.13.5 Fault Recovery in TrueFFS

A fault can occur whenever data is written to flash; thus, for example, a fault can occur:

- In response to a write request from the file system.
- During garbage collection.
- During erase operations.
- During formatting.

TrueFFS can recover from the fault in all cases except when new data is being written to flash for the first time. This new data will be lost. However, once data is safely written to flash, it is essentially immune to power failures. All data already resident in flash is recoverable, and the file and directory structures of the disk are retained. In fact, the negative consequence of a power interruption or fault is the need to restart any incomplete garbage collection operation. This section describes fault occurrence and fault recovery in TrueFFS.

Recovering During a Write Operation

A write or erase operation can fail because of a hardware problem or a power failure. As mentioned in above, TrueFFS uses an "erase after write" algorithm, in which the previous data is not erased until after the update operation has successfully completed. To prevent the possible loss of data, TrueFFS monitors and verifies the success of each write operation, using a register in which the actual written data is read back and compared to the user data. Therefore, a data sector cannot be in a partially written state. If the operation completes, the new sector is valid; if the update fails, the old data is not lost or in any way corrupted.

TrueFFS verifies each write operation, and automatically attempts a second write to a different area of flash after any failure. This ensures the integrity of the data by making failure recovery automatic. This write-error recovery mechanism is especially valuable as the flash medium approaches its cycling limit (end-of-life). At that time, flash write/erase failures become more frequent, but the only user-observed effect is a gradual decline in performance (because of the need for write retries).

Recovering Mapping Information

TrueFFS stores critical mapping information in flash-resident memory, thus it is not lost during an interruption such as a power loss or the removal of the flash medium. TrueFFS does, however, use a RAM-resident mapping table to track the contents of flash memory. When power is restored or the medium reconnected, the the RAM-resident version of the flash mapping table is reconstructed (or verified) from flash-resident information.

NOTE: Mapping information can reside anywhere on the medium.However, each erase unit in flash memory maintains header information at a predictable location. By carefully cross-checking the header information in each erase unit, TrueFFS is able to rebuild or verify a RAM copy of the flash mapping table.

Recovering During Garbage Collection

After a fault, any garbage collection in process at the time of the failure must be restarted. Garbage area is space that is occupied by sections that have been deleted by the host. TrueFFS reclaims garbage space by first moving data from one transfer unit to another, and then erasing the original unit.

If consistent flash failures prevent the necessary write operations to move data, or if it is not possible to erase the old unit, the garbage collection operation fails. To minimize a failure of the write part of the transfer, TrueFFS formats the flash medium to contain more than one transfer unit. Thus, if the write to one transfer unit fails, TrueFFS retries the write using a different transfer unit. If all transfer units fail, the medium no longer accepts new data and becomes a read-only device. This does not, however, have a direct effect on the user data, all of which is already safely stored.

Recovering During Formatting

In some cases, sections of the flash medium are found to be unusable when flash is first formatted. Typically, this occurs because those sections are not erasable. As long as the number of bad units does not exceed the number of transfer units, the medium is considered usable as a whole and can be formatted. The only noticeable adverse effect is the reduced capacity of the formatted flash medium.

9
VxDCOM Applications
COM Support and Optional Component VxDCOM

9.1 Introduction

VxDCOM is the technology that implements COM and distributed COM (DCOM) on VxWorks. The name VxDCOM refers both to this technology and to the optional product. The VxDCOM optional product adds DCOM capabilities to the basic COM support that is included in the standard VxWorks facilities.

COM stands for the Component Object Model, which is a binary specification for component-based object communication. The Wind River VxDCOM technology was designed to significantly facilitate the creation of both COM and DCOM components, by automating much of the boiler-plate code. Thus, VxDCOM enables users to easily write distributed object applications for use in real-time embedded systems software.

The VxDCOM documentation assumes working knowledge of the COM technology, and focuses on the VxDCOM facilities that are used to create server applications for execution on VxWorks. The documentation comprises both this chapter and the *Tornado User's Guide: Building COM and DCOM Applications*. The latter includes a step-by-step overview of how to create and build a VxDCOM application. It covers procedural information for running the VxDCOM wizard, descriptions of the generated output, and the process of building and deploying VxDCOM applications.

This chapter covers the following topics, which are primarily reference material and programming issues:

– a brief overview of the VxDCOM technology

– the Wind Object Template Library (WOTL) classes

– the auto-generated WOTL skeleton code

– the Wind IDL compiler and command-line options

– the structure and meaning of IDL definitions in the auto-generated files

– the configuration parameters for DCOM support in VxWorks

– real-time extensions and OPC interfaces

– tips and examples for writing implementation code

– an implementation comparison of VxDCOM and ATL

The demo example used in this chapter is a DCOM server application that includes both a Visual Basic and a C++ client. Source for this demo is located in the *installDir***/host/src/vxdcom/demo/MathDemo** directory.

9.2 An Overview of COM Technology

COM is a specification for a communication protocol between objects, which are called *COM components*. COM components are the basic building blocks of COM client/server applications. The COM component is the server that provides services to a client application by means of the functionality it advertises as its *COM interfaces*. COM interfaces are sets of method prototypes that, viewed as a whole, describe a coherent, well-defined functionality or service that is offered to COM clients.

9.2.1 COM Components and Software Reusability

COM components are instantiated from classes called *CoClasses*. The CoClass definitions include (single or multiple) inheritance from COM interfaces. The CoClass is then required to implement the methods of the interfaces from which it is derived. While the interface itself strictly defines the service it provides, the implementation details are completely hidden from the client in the CoClass implementation code. Clients are essentially unaware of COM components and interact only with COM interfaces. This is one of the strengths of the COM technology design; and it enables software developers to both update and reuse components without impacting the client applications.

The following sections describe these fundamental elements of COM in more detail.

COM Interfaces

An interface is a named set of pure method prototypes. The interface name typically reflects the functionality of its methods and, by convention, begins with the capital letter I. Thus, **IMalloc** might represent an interface that allocates, frees, and manages memory. Similarly, **ISem** might be an interface that encapsulates the functionality of a semaphore.

As part of the COM technology, basic interface services are defined in the *COM library*. The COM and DCOM libraries that ship with VxDCOM are implementations of the basic interfaces that are required for the aspects of COM technology that VxDCOM supports.

NOTE: These COM and DCOM libraries are used by the C++ template class library (WOTL) and are shipped with VxDCOM. For the API definitions, see **comLib.h** and **dcomLib.h**.

As a developer, you typically define your own custom interfaces. Interface definitions must conform to the COM specification, including descriptive attributes and specifications for the interface, its methods, the method parameters, and the return type (see *The Interface Definition*, p.369). By strictly adhering to this specification, the COM interface becomes a contractual agreement between client and server, and enables the communication protocol to function properly.

NOTE: Interface definitions are pure prototypes, similar to abstract C++ classes that contain only pure virtual methods. In fact, the Wind Object Template Library (WOTL), which you use to write VxDCOM applications, implements interfaces in exactly this manner.

The contract between client and server is to provide a service; but it is not (and need not be) a guarantee how that service is implemented. The implementation details are hidden in the CoClass.

CoClasses

CoClasses are the classes that declare the interfaces and implement the interface methods. The CoClass definition includes a declaration of all the interfaces that it agrees to support. The CoClass implementation code must implement all of the methods of all of the interfaces declared by that CoClass. When the CoClass is instantiated, it becomes a COM component, or server. Client applications query for interfaces services they need, and if your server supports those interfaces, then it communicates with the client application through those interfaces.

Interface Pointers

In the COM model, the communication protocol between COM clients and servers in handled using pointers to the COM interfaces. The interface pointers are used to pass data between COM clients and COM servers. The client application uses COM interface pointers to query COM servers for specific interfaces, to obtain access to those interfaces, and to invoke the methods of the interfaces. Because the COM specification and communication protocol is based on these pointers, it is considered to be a binary standard.

Because COM technology uses a binary standard, it is theoretically possible for COM components to be written in any language, and to run on any operating system, while still being able to communicate. In this way, COM technology offers great flexibility and component reusability to software developers, especially those wanting to port applications to different operating systems.

NOTE: Currently, the only languages supported for COM servers under VxDCOM are C and C++; for DCOM servers, only C++ is currently supported.

VxDCOM Tools

Correctly defining interfaces and CoClasses, according to the COM specification, can be detailed and tedious. However, because the specification follows standard rules, the VxDCOM tools are able to facilitate this process by automatically generating much of the code for you. For example, using the VxDCOM wizard, you simply name your interface, and then select method parameter types and attributes from predefined lists. The wizard automatically generates the correct method return type for you, as well as your interface and CoClass definitions.

When building the application, the Wind IDL (Interface Definition Language) compiler, **widl**, generates additional code used for server registration and marshaling. Thus, you only need to use the tools, write your client and server implementation code, and build your project, in order to develop COM applications.

9.2.2 VxDCOM and Real-time Distributed Technology

COM provides a common framework for describing the behavior and accessibility of software components. Extending the basic COM technology across process and machine boundaries into the realm of distributed objects requires a network RPC protocol. For this, DCOM uses Object RPC (ORPC), a Microsoft extension of the DCE-RPC specification. ORPC uses the marshaled interface pointer as the protocol for communication between COM components, specifying the way references to the component's interfaces are represented, communicated, and maintained. COM provides this functionality in its primary interfaces as defined in the standard COM libraries.

The VxDCOM technology supports the basic COM interfaces as part of the standard VxWorks facilities, and the DCOM network protocol as part of the VxDCOM optional product. These implementations are documented in the VxDCOM COM and DCOM libraries, which contain sets of COM and DCOM-related interfaces targeted for embedded systems development, as well as support for ORPC.

NOTE: For details on these interfaces, see the appropriate **.idl** files, such as **vxidl.idl**. For the API documentation, see **comLib.h** and **dcomLib.h**.

VxDCOM supports both in-process and remote server models. You can write a server on a VxWorks target that provides services to client applications running on other VxWorks targets or on PCs running Windows NT. Using the DCOM protocol for embedded intelligent systems (such as telecommunications devices, industrial controllers, and office peripherals) allows developers to extend the COM programming environment out into the local area network, or even into the Internet, without concern for networking issues.

NOTE: VxDCOM does not support VxWorks to Windows NT when starting the connection up on a VxWorks target. A callback from NT to VxWorks can be used instead. VxDCOM does support VxWorks to NT connections, if all NT security is disabled. For details, see the *Tornado User's Guide: Authenticate the Server* and the *Tornado User's Guide: Registering, Deploying, and Running Your Application*.

9.3 Using the Wind Object Template Library

The Wind Object Template Library (WOTL) is a C++ template class library designed for writing VxDCOM clients and servers. It is source-compatible with a subset of ATL (the Microsoft COM template class library), on which it was modeled. WOTL is the framework you use to write your client and server code.

You run the VxDCOM wizard to create your COM components. This wizard generates output files. Among these output files are headers and implementation source files that contain skeleton code in WOTL. Using this skeleton code, you complete the implementation details for your server, and for any optional client applications.

The *Tornado User's Guide: Building COM and DCOM Applications* describes which files to use to write implementation code appropriate to your type of application. Further details are discussed in *9.10 Writing VxDCOM Servers and Client Applications*, p.377 and *9.10.3 Writing Client Code*, p.380. Code snippets in this chapter are from the **CoMathDemo** demo, located in the *installDir***/host/src/VxDCOM/demo/MathDemo** directory.

9.3.1 WOTL Template Class Categories

There are three VxDCOM template classes that you can define using WOTL. These types are distinguished by whether or not the class uses a CLSID, and whether or not there can be only one instantiation of the class at any one time. A CLSID is a unique identification value for a class, and allows the class to be externally instantiated via a class factory.

NOTE: Class factories are objects that instantiate CoClasses based on CLSIDs.

The three WOTL template classes are categorized as follows:

- **True CoClasses:** These are true COM classes, meaning that they have registered CLSIDs and are instantiated through class factories. The **CComCoClass** template class is used to declare these classes. This is the template class that the VxDCOM wizard uses to generates skeleton code for the CoClass definitions for your COM components.

- **Singleton classes:** These are also true CoClasses, but for which there is only one instance. Thus, every invocation of **IClassFactory::CreateInstance( )** returns the same instance. To declare such a class, use the macro **DECLARE_CLASSFACTORY_SINGLETON** in your class definition.

- **Lightweight classes:** These are simple classes that are not technically true COM classes. The template class has no associated CLSID (class identification value) and thus, instantiates objects by using a default class-factory creation mechanism. These can never be DCOM classes. To declare such a class, use the **CComObject** class. You typically use these classes to create internal objects that enhance the object-oriented functionality of your code.

  The WOTL classes and class templates are described below, with details on how to use them. WOTL classes are declared in the header *installDir***/target/h/comObjLib.h**. Some additional helper classes, such as **VxComBSTR**, are defined in *installDir***/target/h/comObjLibExt.h**. For more information, see *9.11 Comparing VxDCOM and ATL Implementations.*, p.385.

9.3.2 True CoClass Template Classes

The VxDCOM wizard automatically generates the template class definitions for your CoClass, including proper derivation from base classes and from interfaces you want to implement in your CoClass. The following sections explain this code.

CComObjectRoot – IUnknown Implementation Support Class

CComObjectRoot is the base class for all VxDCOM classes and provides task-safe **IUnknown** implementation, plus support for aggregatable object. To declare a class that provides concrete implementations for one or more COM interfaces, the class must inherit from both **CComObjectRoot** and the interfaces it implements. The following example declares a class, **CExample**, that supports the interface **IExample** and will implement its methods:

```
class CExample: public CComObjectRoot, public IExample
    {
        // Private instance data
    public:
        // Implementation, including IExample methods...
    };
```

As the implementation class for **IUnknown**, all WOTL implementation classes that you declare should include **CComObjectRoot** as an intimate base class.

CComCoClass – CoClass Class Template

CComCoClass is the template class that provides the CoClass implementation; that is, it includes a publicly-known CLSID for the class, and one or more publicly known interfaces. Objects created from this class are considered to be true COM objects because they can be 'externally' instantiated using the CLSID and calling the COM or DCOM API creation functions **CoCreateInstance( )** and **CoCreateInstanceEx( )**.

CComCoClass wraps the functionality required by the class factory class and the registration mechanism, so that classes derived from it inherit that functionality. **CComClassFactory** is the class factory class template that implements the standard **IClassFactory** COM interface. This interface allows objects to be created at run-time using CLSIDs. The declaration for this class follows the standard WOTL format, inheriting from **CComObjectRoot** and the interface being implemented.

CoClass Definitions

The definition for CoClasses is automatically generated by the wizard. These CoClasses are derived from all of the following:

- the ultimate WOTL base class, **CComObjectRoot**
- the WOTL base class template for all CoClasses, **CComCoClass**
- the primary interface, which is implemented by the CoClass
- all additional interfaces implemented by the CoClass

Example Definition

The example below shows this inheritance in the definition of the server CoClass in the **CoMathDemoImpl.h** header file:

```
class CoMathDemoImpl
    : public CComObjectRoot,
      public CComCoClass<CoMathDemoImpl, &CLSID_CoMathDemo>
      ,public IMathDemo
      ,public IEvalDemo
    {
        // body of definition
    };
```

The generated server implementation header for your application will include a declaration for your CoClass that is similarly derived from **CComObjectRoot**, from **CComCoClass**, your primary interface, and from any additional interfaces. Note that WOTL supports multiple inheritance, which includes inheritance from multiple interfaces; so your CoClass definition derives from each of the interfaces it implements.

Using CLSID Instantiation

The **CComCoClass** class template creates its class instantiation based upon the name of the CoClass and the CLSID. The **&CLSID_CoMathDemo** parameter, in the CoMathDemo example code above, represents the GUID (globally unique identifier) that identifies the CoClass. The CLSID for your CoClass is *CLSID_basename*, and can be found in the *basename*_**i.c** file, generated by the **widl** compilation. The parameter you would use is thus **&CLSID_***basename*.

NOTE: The *CLSID_basename* (class ID) is declared as a **const**, and represents the GUID generated for the CoClass from the compilation of the **.idl** file. *CLSID_basename* is used in the server header and implementation files, and also in the client implementation file. Use this **const** when referencing the CLSID (class ID) for your CoClass.

9.3.3 Lightweight Object Class Template

Lightweight classes create COM objects without a CLSID; and because of this, they are not considered to be true CoClasses. Lightweight classes are typically used internally to optimize the object-oriented design of an application through the use of interfaces.

CComObject is the lightweight object class template for WOTL. **CComObject** is a wrapper template class used to create classes for lightweight objects. The actual implementation class used as an argument to the templatized class **CComObject** derives from **CComObjectRoot** in order to inherit **IUnknown** interface support.

In the following **CComObject** template class instantiation, **CExample** is the class defined in the example above. It derives from **CComObjectRoot** and implements the **IExample** interface:

```
CComObject<CExample>
```

Lightweight classes do not have a CLSID and, thus, cannot be externally instantiated using the normal class-factory method. For this reason, **CComObject** provides default class-factory implementation. These classes are instantiated by calling the function **CComObject<CExample>::CreateInstance( )**, rather than by using **CoCreateInstance( )**. This function takes the same arguments as **IClassFactory::CreateInstance( )**.

The VxDCOM wizard does not generate definitions for this template class. To create a lightweight class object, simply add the definition and implementation code to your header and source files.

9.3.4 Single Instance Class Macro

To define a class as a singleton, that is, a class for which there is only one instance, you include a **DECLARE_CLASSFACTORY_SINGLETON** statement in your class definition.

```
class CExample
    : public CComObjectRoot,
      public CComCoClass<CExample, &CLSID_Example>,
      public IExample
    {
        // Private instance data
    public:
        DECLARE_CLASSFACTORY_SINGLETON();
        // Implementation, including IExample methods...
    };
```

When you declare a class using this macro, every call to **IClassFactory::CreateInstance( )** returns the same instance. The VxDCOM wizard does not generate definitions for this template class. To create a single instance class object, simply add the definition and implementation code to your header and source files.

9.4 Reading WOTL-Generated Code

The purpose of this section is to familiarize you with the code appearing in the wizard-generated output files. For the most part, you will not need to modify this code to implement your server or client. However, WOTL includes many helper macros, which have been defined for convenience and readability. This section discusses these macro definitions to enable users to more easily read the generated WOTL code and, over time, better understand how VxDCOM implements the COM technology. This section also summarizes the generated interface and CoClass definitions and the files they appear in.

9.4.1 WOTL CoClass Definitions

The definition for your server CoClass is generated by the wizard in the implementation header, *basename***Impl.h**. This header includes two basic header files:

- the WOTL header file, *installDir***/target/h/comObjLib.h**
- the **widl**-generated interface method prototype header, *basename***.h**.

You can include other header files as necessary, depending upon your implementation code. For example, the **CoMathDemoImpl.h** header file looks like this:

```
/* CoMathDemoImpl.h -- auto-generated COM class header */

#include "comObjLib.h"    // COM-object template lib
#include "CoMathDemo.h"   // IDL-output interface defs
#include <string>
```

The **widl**-generated interface prototype header, *basename***.h** (mentioned above), declares your interfaces as abstract C++ classes with only pure virtual functions. Those methods are then re-defined, in the CoClass that implements them, as **virtual STDCALL** functions that return an **HRESULT**. Essentially, the CoClass is defined as implementing the virtual methods of its purely abstract base class, which is the interface.

9.4.2 Macro Definitions Used in Generated Files

There are three automatically generated WOTL files that use macros in their definitions. These files are:

- the interface prototype header, *basename***.h**, generated by **widl**
- the server CoClass header, *basename***Impl.h,** generated by the wizard
- the server implementation file, *basename***Impl.cpp**, generated by the wizard

The macros used in these files are defined in the header *installDir***\target\h\comLib.h** file. The five definitions are as follows:

```
#define STDMETHODCALLTYPE      STDCALL
#define STDMETHODIMP           HRESULT STDMETHODCALLTYPE
#define STDMETHODIMP_(type)    type STDMETHODCALLTYPE
#define STDMETHOD(method)      virtual HRESULT STDMETHODCALLTYPE method
#define STDMETHOD_(type,method)   virtual type STDMETHODCALLTYPE method
```

These macros are used when mapping interface method definitions across **.idl** files, header files, and implementation files. The details of this process are described in the sections that follow. Table 9-1 provides a quick reference summary of these mappings when reading the files.

Table 9-1 **Interface Method Definition Mappings Across Files**

File	Meaning	Tool Used for Code Generation	Method Prototype Syntax
basename.**idl**	IDL interface definitions	wizard	**HRESULT** *routine_name* **(***parameters***);**
basename.**h**	C++ interface method prototypes	**widl**	**virtual HRESULT STDMETHODCALLTYPE** *routine_name***(***parameters***) = 0;**
*basename***Impl.h**	C++ CoClass method prototypes	wizard	**STDMETHOD(***method***) (***parameters***);**
*basename***Impl.cpp**	C++ CoClass method definitions	wizard	**STDMETHODIMP** *coclass* **::** *routine_name* **(***params***) { // implementation code }**

Mapping IDL Definitions to Interface Header Prototypes

Using these macros, the interface method prototypes are mapped so that each interface method definition, found in the **.idl** file, is defined as a **virtual STDCALL** method that returns an **HRESULT** in the generated interface header file, *basename*.**h**. Thus, the following definition syntax in the **.idl** file:

```
HRESULT routine_name (parameters);
```

becomes:

```
virtual HRESULT STDMETHODCALLTYPE routine_name(parameters) = 0;
```

in the *basename*.**h** header file.

Mapping Interface Prototypes to CoClass Method Definitions

Further, the macro **STDMETHODCALLTYPE** is defined in **comBase.h** as **STDMETHOD:**

```
#define STDMETHOD(method)    virtual HRESULT STDMETHODCALLTYPE method
```

So, for each interface method you define in the wizard, the final prototype in your *basename***Impl.h** file is:

```
STDMETHOD(method) (parameters);
```

NOTE: If you edit the C++ header, *basename***Impl.h**, by hand, use the **CoMathDemo** files or other wizard generated files as examples.

Defining CoClass Methods in Implementation Files

The server implementation file uses the macro **STDMETHODIMP** to represent a **STDCALL** method that returns an **HRESULT**. Therefore, the method definitions in your generated implementation files, as in the **CoMathDemoImpl.cpp** file, are of the following form:

```
STDMETHODIMP coclass :: routine_name (params) { // implementation code }
```

9.4.3 Interface Maps

WOTL uses the **COM_MAP** style of interface mapping within the class definition. These macros are similar to those used in ATL and are part of the implementation of the **IUnknown** method, **QueryInterface( )**. The WOTL library definitions of **COM_MAP** macros are found in the WOTL header file, **comObjLib.h**.

The **COM_MAP** macros define a function called **_qi_impl( ),** which does run-time casts to obtain the requested interface pointer. Other than that, the layout of the **COM_MAP** in your CoClass is identical to that of an ATL map; however, only the entries for **COM_INTERFACE_ENTRY** and **COM_INTERFACE_ENTRY_IID** are supported.

The **CoMathDemoImpl.h** header includes an interface map at the end of the public definition of methods. Both interfaces implemented by the CoClass are defined in the interface map:

```
// COM Interface map
BEGIN_COM_MAP(CoMathDemoImpl)
    COM_INTERFACE_ENTRY(IMathDemo)
    COM_INTERFACE_ENTRY(IEvalDemo)
END_COM_MAP()
```

The header files generated for your CoClass will have similar interface map definitions for your CoClass interfaces.[1]

[1] The default **IUnknown** is always the first **COM_INTERFACE_ENTRY** in the table. Therefore, by definition, this entry must be derived from **IUnknown** or the **static_cast( )** will fail.

9.5 Configuring DCOM Properties' Parameters

If you add DCOM support, you can also configure the DCOM properties' parameters. The meaning and purpose of these parameters are described below, along with the range of possible values and the default setting for each.

For a description of how to change these parameters, see the *Tornado User's Guide: Building COM and DCOM Client and Server Applications*.

- **VXDCOM_AUTHN_LEVEL** Specifies the level of authentication required by the application.

  Values

  0 = default, no authentication required

  1 = no authentication required

  2 = **RPC_CN_AUTHN_LEVEL_CONNECT**, which means that the application must create a userId and password combination (using the **vxdcomUserAdd( )** API) so that incoming connections can be authenticated.

  Default

  0

- **VXDCOM_BSTR_POLICY** Sets configuration for marshaling of **BSTR**s.

  Values

  TRUE = **BSTR**s are marshaled as counted strings (byte-arrays), in which the first byte specifies the length of the string and the remaining bytes are represented as ASCII characters.

  FALSE = **BSTR**s are marshaled as **BSTR**s, that is as wide-character strings, which are two-byte unicode character strings.

  Default

  FALSE

- **VXDCOM_CLIENT_PRIORITY_PROPAGATION** Adds priority schemes and real-time extension priority configuration to basic DCOM functionality.

  Values

  TRUE = propagates the client priority, enabling the server to run at the same priority as the client.

  FALSE = server runs at its own priority, independent of the client priority.

Default
TRUE

For more information, see *9.8.2 Configuring Client Priority Propagation on Windows*, p.375 .

- **VXDCOM_DYNAMIC_THREADS** Specifies the number of additional threads that can be allocated at peak times.

  Values
  Range is 0..32

  Default
  30

- **VXDCOM_OBJECT_EXPORTER_PORT_NUMBER** Sets the configuration for validating the **ObjectExporter** port number after a system reboot. If this parameter is set to zero, the port number is assigned dynamically.

  Values
  Range is 1025..65535 (**short int** values)

  Default
  65000

- **VXDCOM_SCM_STACK_SIZE** Specifies the stack size, at build time, of the Service Control Manager (SCM) task.This parameter is mainly used to configure PPC60x target architectures, which can create a larger stack frame than other standard architectures.

  On most architectures, the default value can be used.

  Default
  30K

  If a stack exception or data exception is seen in the **tSCM** task, use the browser to check whether the stack has been exhausted. If it has, then increase this value.

- **VXDCOM_STACK_SIZE** Specifies the stack size of threads in the server threadpool.

  Values
  Recommended range is 16K...128K

  Default
  16K

If data exceptions or stack exceptions are seen in **tCOM** tasks, use the browser to check whether stack exhaustion is the cause. If it is, increase this value.

- **VXDCOM_STATIC_THREADS** Specifies the number of threads to preallocate in the server threadpool.

  Values
  Range is 1..32

  Default
  5

  VxDCOM starts up a number of initialized threads to speed up CoClass startup times. Set this value to the average number of CoClasses running within the system in order to speed up CoClass initialization.

- **VXDCOM_THREAD_PRIORITY** Specifies the default priority of threads in the server threadpool. For more information, see *9.8.3 Using Threadpools*, p.376.

  Values
  Same range as that of the VxWorks task priority.

  Default
  150

9.6 Using the Wind IDL Compiler

The Wind IDL Compiler, **widl**, is a command-line tool that is integrated into the build process in the Tornado IDE. Thus, when you build a D/COM project, **widl** is run automatically.

9.6.1 Command-Line Syntax

You can run **widl** from the command line; however, doing so becomes redundant once the **.idl** file is part of the project. The command-line syntax for running **widl** is:

```
widl [-IincDir] [-noh] [-nops] [-dep] [-o outPutDir] idlFile[.idl]
```

The *idlFile* must be specified, with or without the **.idl** extension, which is assumed. The other command-line switches are optional and are described below:

-I *incDir*
Add specified include directory to the header search path.

-noh
Do not generate header file.

-nops
Do not generate proxy/stub file.

-dep
Generate GNU-style dependency file.

-o *outputDir*
Write output from **widl** to the specified output directory.

9.6.2 Generated Code

When **widl** compiles the wizard-generated **.idl** file, additional code is generated that is added to these three files which were created by the wizard with empty contents: *basename*_**i.c**, *basename*.**h**, and *basename*_**ps.cpp**.

basename.**tlb** (DCOM only)
Type library used to register the server in the Windows Registry (generated by MIDL when the **nmakefile** is run to build the Windows DCOM client program, and required for DCOM projects only).

*basename*_**ps.cpp** (DCOM only)
Proxy/stub code required for marshalling interface method parameters between the client and the server.

basename.**h**
Interface prototypes based on the interface definitions in the **.idl** file output from the VxDCOM wizard.

*basename*_**i.c**
Associations of GUIDs (unique identification numbers) with the elements of the **.idl** file (for example, each interface has an associated IID or interface identifier, each CoClass has a unique CLSID or class identifier, and so on).

WARNING: The **widl** tool only generates proxy/stub code for the interfaces defined in the IDL file being compiled, and not for interfaces defined in imported IDL files. To generate the proxy/stub code for those interfaces, you must separately compile each IDL file that defines each of the required interfaces. The exception to this rule is the IDL file, *installDir***\target\h\vxidl.idl**, whose proxy/stub code routines are supplied in **DCOM** support components.

9.6.3 Data Types

The **widl** tool compiles both automation data types and non-automation data types. The specific individual types from these two groups that **widl** compiles with this version of VxDCOM are listed below.

Automation Data Types

Automation data types are simple types, interface pointers, **VARIANT**s, and **BSTR**s as described below.

long, long *, int, int *, short, short *, char, char *, float, float *, double, double *
Integral values must be explicitly declared as one of the above or a pointer type of one of the above.

IUnknown *, IUnknown **
Includes interface pointer types.

BSTR, **BSTR ***
BSTRs are wide-characters, which are 16-bit unsigned integers, used to create character strings. Because COM APIs take wide-character strings as parameters, strings are passed as **BSTRs**. You can modify the method of reading and passing BSTRs in the configuration options parameters for DCOM.

If you need to convert between the standard 8-bit ASCII type and the wide-character strings, you can use the following routines:

comWideToAscii( )
Convert a wide-string to ASCII.

comAsciiToWide( )
Convert an ASCII string to a wide-string.

NOTE: VxDCOM also supports the ATL macros, **OLE2T** and **T2OLE**. However, these macros call **alloca( )**, which uses memory from the current stack, meaning the macros could consume an entire stack space if used inside a loop. It is recommended that you use the **VxComBSTR** class, defined in **comObjLibExt.h**, as an alternative. For details, see *9.11.7 VxComBSTR*, p.391.

VARIANT, **VARIANT ***
Includes all values of the **VARTYPE** enumeration that are valid and supported by VxDCOM in a **VARIANT**, including **SAFEARRAY**. For details on **SAFEARRAY** support, see *SAFEARRAY with VARIANTS*, p.364.

Non-Automation Data Types

The non-automation types that **widl** can correctly compile are:

simple array
an array with a fixed, compile-time constant size

fixed-size structure
a structure whose members are all either automation-types, simple arrays, or other fixed-size structures

conformant array
an array whose size is determined at run time by another parameter, structure field, or expression, using the **size_is** attribute

conformant structure
a structure of variable size, whose final member is a conformant array, of size determined by another structure member

string
a pointer to a valid character type (single or double-byte type) with the **[string]** attribute

⚠ **CAUTION:** **widl** does not support the type **union**.

If you use a non-automated type, then you will need to build a proxy DLL. You can use **midl** to generate the DLL for you, and then you need to build it. For details on registering proxy DLLs, see the *Tornado User's Guide: Register Proxy DLLs on Windows*.

If you are using non-automation data types because you are using OPC interfaces, then you need to also add the **DCOM_OPC** support component to your system, and install the OPC Proxy DLL on windows. For more information, see the *Tornado User's Guide: OPC Program Support*.

SAFEARRAY with VARIANTS

VxDCOM support for **SAFEARRAY**s is only available when marshaling within a **VARIANT**, it cannot be marshaled otherwise. VxDCOM implementation of **SAFEARRAY** supports both the use of COM based **SAFEARRAY**s and also marshalling **SAFEARRAY**s from DCOM server to DCOM client. When using **SAFEARRAY**s in your application, note the following documented sections on specific support and restrictions for both COM and DCOM application.

COM Support

When using **SAFEARRAY**s under COM, the following is a list of supported features.

- Support for Multi Dimension **SAFEARRAY**s of the following types:

  VT_UI1
  VT_I2
  VT_I4
  VT_R4
  VT_R8
  VT_ERROR
  VT_CY
  VT_DATE
  VT_BOOL
  VT_UNKNOWN
  VT_VARIANT
  VT_BSTR

- Support for **SAFEARRAY**s of **VARIANT**s in which the **VARIANT**s can contain **SAFEARRAY**s.

- Support for a minimal **SAFEARRAY** API.

⚠ **CAUTION:** The memory structure returned by **SafeArrayAccessData( )** is not guaranteed to be the same layout as that returned by the Microsoft API.

DCOM Support

When using **SAFEARRAY**s under DCOM, the following is a list of supported features and restrictions.

- Support for Single Dimension **SAFEARRAY**s of the following types:

  VT_UI1
  VT_I2
  VT_I4
  VT_R4
  VT_R8
  VT_ERROR
  VT_CY
  VT_DATE
  VT_BOOL
  VT_BSTR

- VxDCOM only supports **SAFEARRAY**s less than 16Kb in size.
- Because Microsoft DCOM fragments packets that are over 4 KB in length, you should only send **SAFEARRAY**s of 4 KB or less in size when sending a **SAFEARRAY** from a Microsoft platform.

HRESULT Return Values

All interface methods require an **HRESULT** return type. **HRESULT**s are the 32-bit return value from all ORPC methods and are used to handle RPC exceptions. By using ORPC and **HRESULT** return types the COM technology can provide a virtually transparent process of object communication from the developer's perspective. This is because, as long as client code returns an **HRESULT**, the client application can access all objects, whether in-process or remote, in a uniform transparent fashion. In fact, in-process servers communicate with client applications by using C++ virtual method calls; whereas remote servers communicate with client applications by using proxy/stub code and by invoking local RPC services. However, this process is all transparent to the programmer, because it happens automatically within the COM mechanism. The only difference is that making a remote procedure call requires more overhead.[2]

The VxDCOM wizard automatically generates an **HRESULT** return type for all methods you define. When writing your implementation code, you can refer to the

[2] As there is no need to support cross-process (and local RPC) server models, VxDCOM supports only the in-process and remote server models.

header file *installDir***\target\h\comErr.h** for a comprehensive list of **HRESULT** error constants. Table 9-2 lists the most commonly used **HRESULT** values.

Table 9-2 **Common HRESULT Values**

Value	Meaning
S_OK	Success. (0x00000000)
E_OUTOFMEMORY	Insufficient memory to complete the call. (0x80000002)
E_INVALIDARG	One or more arguments are invalid. (0x80000003)
E_NOINTERFACE	No such interface supported. (0x80000004)
E_ACCESSDENIED	A secured operation has failed due to inadequate security privileges. (0x80070005)
E_UNEXPECTED	Unknown, but relatively catastrophic error. (0x8000FFFF)
S_FALSE	False. (0x00000001)

9.7 Reading IDL Files

IDL (Interface Definition Language) is a specification used by COM for defining its elements (interfaces, methods, type libraries, CoClasses, and so on). The file containing these definitions is written in IDL and, by convention, is identified by an **.idl** extension. VxDCOM assumes this extension for the tools to work properly. Therefore, the wizard generated interface definition file has this extension.

9.7.1 IDL File Structure

The **.idl** file structure includes the interface, CoClass, and type-library definitions, each of which contains a header and a body. The CoClass definition is itself part of the library body. This structure is fairly standard, although you may encounter some **.idl** files that diverge slightly from it.

Below is an example of the typical syntax for a **.idl** file with multiple interface definitions, where the library definition contains the CoClass definition in its body; and the CoClass definition contains the interfaces it implements in its body.

```
//import directives, typedefs, constant declarations, other definitions

[attributes] interface interfacename: base-interface {definitions};
[attributes] interface interfacename: base-interface {definitions};

//additional interface definitions as required...

[attributes] library libname {definitions};
```

The sample **.idl** file below is from the CoMathDemo demo program found in the *installDir***/host/src/VxDCOM/MathDemo** directory. It demonstrates the structure of an **.idl** file that defines two interfaces, **IMathDemo** and **IEvalDemo**, a type library, **CoMathDemoLib**, and a CoClass, **CoMathDemo,** that implements both interfaces.

```
#ifdef _WIN32
import "unknwn.idl";
#else
import "vxidl.idl";
#endif

[
    object,
    oleautomation,
    uuid(A972BFBE-B4A9-11D3-80B6-00C04FA12C4A),
    pointer_default(unique)
]
interface IMathDemo:IUnknown
    {
    HRESULT pi([out,retval]double* value);
    HRESULT acos ([in]double x, [out,retval]double* value);
    HRESULT asin ([in]double x, [out,retval]double* value);
    HRESULT atan([in]double x, [out,retval]double* value);
    HRESULT cos([in]double x, [out,retval]double* value);
    HRESULT cosh([in]double x, [out,retval]double* value);
    HRESULT exp([in]double x, [out,retval]double* value);
    HRESULT fabs([in]double x, [out,retval]double* value);
    HRESULT floor ([in]double x, [out,retval]double* value);
    HRESULT fmod([in]double x,[in]double y,[out,retval]double* value);
    HRESULT log ([in]double x, [out,retval]double* value);
    HRESULT log10 ([in]double x, [out,retval]double* value);
    HRESULT pow ([in]double x,[in]double y,[out,retval]double* value);
    HRESULT sin ([in]double x, [out,retval]double* value);
    HRESULT sincos ([in]double x,
                    [out]double* sinValue,
                    [out]double* cosValue);
    HRESULT sinh ([in]double x, [out,retval]double* value);
    HRESULT sqrt ([in]double x, [out,retval]double* value);
    HRESULT tan ([in]double x, [out,retval]double* value);
    HRESULT tanh ([in]double x, [out,retval]double* value);
    };
```

```
[
    object,
    oleautomation,
    uuid(4866C2E0-B6E0-11D3-80B7-00C04FA12C4A),
    pointer_default(unique)
]
interface IEvalDemo:IUnknown
    {
    HRESULT eval ([in]BSTR str, [out,retval]double* value);
    HRESULT evalSubst ([in]BSTR str,
                       [in]double x,
                       [out,retval] double* value);
    };

[
    uuid(A972BFC0-B4A9-11D3-80B6-00C04FA12C4A),
    version(1.0),
    helpstring("CoMathDemo Type Library")
]

library CoMathDemoLib
    {
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(A972BFBF-B4A9-11D3-80B6-00C04FA12C4A),
        helpstring("CoMathDemo Class")
    ]
    coclass CoMathDemo
        {
        [default] interface IEvalDemo;
        interface IMathDemo;
        };
};
```

The following sections describe the syntax for the parts of an **.idl** file.

The import Directive

The **import** directive precedes the interface definition and specifies another **.idl** or header file. These imported files contain definitions - such as typedefs, constant declarations, and interface definitions - that can be referenced in the importing IDL file. All interfaces inherit from the **IUnknown** interface; therefore all interface definitions will conditionally include either the WIN32 header, **unknwn.idl**, or **vxidl.idl**, which implements **IUnknown** under VxDCOM when running on a target.

The Interface Definition

The *interface definition* in the **.idl** file specifies the actual contract between the client application and server object. It describes the characteristics of each interface in an interface header and an interface body. The syntax for an interface definition is:

[*attributes***] interface** *interfacename***:** *base-interface* **{***definitions***};**

Interface Header

The *interface header* is the section at the beginning of the interface definition. The interface header comprises both the information that is enclosed in square brackets, along with the keyword **interface**, followed by the interface name. The information within the brackets is an attribute list describing characteristics that apply to the interface as a whole. This information is global to the entire interface (in contrast to attributes applied to interface methods.) The attributes describing the interface - **[object]**, **[oleautomation]**, **[uuid]**, and **[pointer-default]** - are the standard generated attributes for VxDCOM interface definitions and are discussed in *9.7.2 Definition Attributes*, p.370.

Interface Body

The *interface body* is the section of the interface definition that is enclosed in C-style braces ({ }). This section contains remote data types and method prototypes for the interface. It can optionally include zero or more import lists, pragmas, constant declarations, general declarations, and function declarators.

Library and CoClass Definitions

The type library and CoClass definitions follow the same syntax pattern as that of an interface definition. The *library definition* syntax is:

[*attributes***] library** *libname* **{***definitions***};**

The library name is preceded by descriptive attributes and followed by a body of definitions that are enclosed in C-style braces ({ }). The **library** keyword indicates that the compiler should generate a type library. The type library includes definitions for every element inside of the library block, plus definitions for any elements that are defined outside, and referenced from within, the library block. The *CoClass definition* lies within the {*definitions*} or body section of the library block, and has its own header and body sections.

The **[version]** attribute identifies a particular version among multiple versions of the interface. The **[version]** attribute ensures that only compatible versions of client and server software will be connected.

The **[helpstring]** attribute specifies a zero-terminated string of characters containing help text that is used to describe the element to which it applies, in this example, the type library. The **[helpstring]** attribute can be used with—a type library, an import library, an interface, a module—or with a CoClass statement, typedefs, properties, and methods.

The CoClass statement is used to define the CoClass and the interfaces that it supports. The CoClass definition is similar to the interface definition. It is comprised of a set of attributes, which requires the **[uuid]** attribute (representing the CLSID), the **CoClass** keyword, the CoClass name, and a body of definitions. All of the interfaces the CoClass implements are listed in the CoClass body; thus, you must add any additional interfaces that your CoClass implements to that list.[3] You will also need to add those interfaces to the CoClass definition in the server implementation header file, as described in *9.10 Writing VxDCOM Servers and Client Applications*, p.377.

9.7.2 Definition Attributes

Interface attributes are annotations that specify certain qualities of your interface. Interface attributes are grouped together in a comma-delimited list, surrounded by brackets. The attribute list always precedes whatever object the attributes in the list are describing. Attributes can also be applied to the libraries and CoClasses. However, some attributes, or combinations of attributes, are valid for one definition type and not another (see *Attribute Restrictions for VxDCOM*, p.372 and *Library and CoClass Definitions*, p.369). For comprehensive information on attributes, see the Microsoft COM specification.

IDL File Attributes

When you run the D/COM Application Wizard, the generated interface definition contains the following attributes as defaults. If you intend to modify any of them, you must follow the language restrictions. (see *Attribute Restrictions for VxDCOM*, p.372.)

[3] This list specifies the full set of interfaces that the CoClass implements, both incoming and outgoing.

[object]

The **[object]** attribute is an IDL extension that specifies the interface as a COM interface (rather than an RPC interface). This attribute tells the IDL compiler to generate all of the proxy/stub code specifically for a COM interface, and to generate a type library for each library block defined within the **.idl** file (see *Library and CoClass Definitions*, p.369). All VxDCOM interface definitions should specify this attribute in the definition.

[oleautomation]

The **[oleautomation]** attribute indicates that the interface is compatible with Automation. VxDCOM interfaces definitions generated by the wizard are declared with this attribute, which specifies that the parameters and return types are automation types.

[pointer_default]

The **[pointer_default]** attribute specifies the default pointer attribute for all pointers except pointers that appear as top-level parameters, such as individual pointers used as function parameters; these automatically default to **ref** pointers. The syntax for the **[pointer_default]** attribute is:

```
pointer_default (ptr | ref | unique)
```

where **ptr**, **ref**, or **unique** can be specified:

ptr
Designates a pointer as a full pointer, with all the capabilities of a C-language pointer, including aliasing.

ref
Designates a reference pointer, one that simply provides the address of data. Reference pointers can never be **NULL**.

unique
Allows a pointer to be **NULL**, but does not support aliasing.

The **[pointer_default]** attribute can apply to pointers returned by functions. For pointers that appear as top-level parameters, such as individual pointers used as function parameters, you must supply the appropriate pointer attribute. For example:

```
HRESULT InterfaceMethod( [unique] VXTYPE* ptrVXTYPE );
```

In this case, the pointer attribute will override the default **[pointer_default]** attribute that appears in the interface header. The **[pointer_default]** attribute is optional in the **.idl** file, and is required only when a function returns an undefined pointer type or when a function contains a parameter with more than one asterisk (*).

[uuid]

The **[uuid]** interface attribute designates a universally unique identifier (UUID) that is assigned to the interface and that distinguishes it from other interfaces. COM technology relies upon these unique values as a means of identifying components and interfaces, as well as sub-objects within the COM system such as interface pointers and type libraries. As part of the **.idl** file, the wizard generates a UUID value for each interface, for the type library, and for the CoClass definition. For a COM interface—that is, for an interface identified by the **[object]** interface attribute—the **[uuid]** attribute is required to determine whether the client can bind to the server. It is used to differentiate versions of public interfaces, so that different vendors can introduce distinct new features without risking compatibility conflicts.

NOTE: A UUID designates a 128-bit value that is guaranteed to be unique. The actual value may represent a GUID, a CLSID, or an IID.

Attribute Restrictions for VxDCOM

The only interface attributes auto-generated for VxDCOM interface definitions are **[object]**, **[oleautomation]**, **[pointer-default]**, and **[uuid]**. Restrictions on using interface attributes with VxDCOM are summarized as follows:

- VxDCOM does not support all interface types; thus, any attributes that are defined for non-supported interface types would not be applicable. For example, because VxDCOM does not support **IDispatch**, the **[dual]** attribute cannot be used.
- Some combinations of attributes are inherently prohibited; for example, the **[version]** attribute cannot be used for a COM interface. The **[version]** attribute, identifies a particular version among multiple versions of an RPC interface. The MIDL compiler does not support multiple versions of a COM interface, so the **[object]** attribute (which specifies the interface as a COM interface) cannot also include a **[version]** attribute.

For a complete list of interface attributes, their meanings, and valid combinations, see the Microsoft documentation.

NOTE: To create a new version of an existing COM interface, use interface inheritance. A derived COM interface has a different UUID but inherits the interface member functions, status codes, and interface attributes of the base interface.

Directional Attributes for Interface Method Parameters

Directional attributes on interface method parameters indicate the direction that data values are passed between methods. The two primary directional attributes are **[in]** and **[out]**. These are the four combinations available for a parameter attribute:

[in]

The **[in]** attribute specifies that the parameter is being passed from caller to callee; that is, the data is being sent only by the caller, for example, the client (caller) to the server (callee).

[out]

The **[out]** attribute specifies that the parameter is being passed from the callee back to the caller, in this case from the server to the client. Because data is passed back through **[out]** parameters, they must always be pointers.

[in,out]

The **[in,out]** attribute lets you apply both **[in]** and **[out]** to a single parameter. Using both directional attributes on one parameter specifies that the parameter is copied in both directions. Such data, that is sent to be modified and passed back to the caller, is passed in a pointer to the resulting data location. When a parameter is defined with this combination of attributes, it increases the overhead of the call.

[out,retval]

The **[retval]** attribute combined with the **[out]** attribute identifies the parameter value as the return value from the function. This attribute is a necessary option because the return type for all COM interface methods must be an **HRESULT**. As a consequence, for some language applications, such as Visual Basic, you need to provide an additional return value. For more information on **HRESULT** return values, see *HRESULT Return Values*, p.365.

9.8 Adding Real-Time Extensions

For DCOM server applications, VxDCOM offers real-time extensions that can improve the performance of your application. Real-time extensions are priority schemes and threadpooling, and they are described in the following sections.

9.8.1 Using Priority Schemes on VxWorks

Priority schemes are used to control the scheduling of server objects running on a VxWorks target. You can specify the priority scheme to use at class registration time in the parameters to the **AUTOREGISTER_COCLASS** macro. The second and third arguments to this macro specify priority scheme details. An example of the **AUTOREGISTER_COCLASS** macro, that is auto-generated by the wizard for the **MathDemo** is:

```
AUTOREGISTER_COCLASS (CoMathDemoImpl, PS_DEFAULT, 0);
```

The generated definition in your server implementation file would look similar with the first argument reflecting your server name.

Second Parameter Priority Scheme

The second argument to **AUTOREGISTER_COCLASS** specifies the priority scheme to be used. The three possible values for this argument are:

PS_SVR_ASSIGNED
This value indicates that the server will always run at a given priority. This priority is assigned on a per object basis at object registration time, for example, as a parameter to the **AUTOREGISTER_COCLASS** macro.

PS_CLNT_PROPAGATED
This value indicates that the worker thread serving the remote invocation should run at the priority of client issuing the request. This effectively provides similar semantics to invoking an operation on the same thread on the same node.

PS_DEFAULT
This value indicates that a default priority can be specified that will be applied to worker threads when the incoming request does not contain a propagated priority. In the **PS_CLNT_PROPAGATED** case a value of -1 indicates that the default server priority (configured in project facility) should be used when the **VXDCOM_ORPC_EXTENT** is not present).

Third Parameter Priority Level

The third argument specifies the actual priority level to assign to either the server, in the case of a **PS_SVR_ASSIGNED** scheme, or to the client, in the case of a **PS_CLNT_PROPAGATED** scheme. In addition, it specifies when the **VXDCOM_ORPC_EXTENT** (which contains the priority) is not present in the request.

If the client is a VxWorks target, the client's priority is conveyed with the RPC invocation to the server, using an ORPC extension mechanism specified in the DCOM protocol definition. This causes the server to execute the method at the same priority as the client task that invoked it.

9.8.2 Configuring Client Priority Propagation on Windows

When configuring VxDCOM projects, the client priority is automatically and transparently transmitted. The client priority propagation can be turned off (see *9.5 Configuring DCOM Properties' Parameters*, p.358), thereby saving a few bytes per request.

Using Windows, you must create a channel hook **IChannelHook** interface to propagate the priority, that is, to add a priority (in the form of an **ORPC_EVENT**) to an outgoing request. It is still the user's responsibility to convert the priority value from the other operating system to an appropriate VxWorks priority. Only the **ClientGetSize( )** and **ClientFillBuffer( )** functions must be implemented. The other **IChannelHook** interface methods, **ClientNotify( )**, **ServerNotify( )**, **ServerGetSize( )**, and **ServerFillBuffer( )** can be empty functions. The following example code implements the **ClientGetSize( )** and **ClientFillBuffer( )** methods:

```
void CChannelHook::ClientGetSize
    (REFGUID uExtent, REFIID riid, ULONG* pDataSize)
    {
    if(uExtent == GUID_VXDCOM_EXTENT)
        *pDataSize = sizeof(VXDCOMEXTENT);
    }

void CChannelHook::ClientFillBuffer
    (REFGUID uExtent, REFIID riid, ULONG* pDataSize, void* pDataBuffer)
    {
    if(uExtent == GUID_VXDCOM_EXTENT)
        {
        VXDCOMEXTENT *data = (VXDCOMEXTENT*)pDataBuffer;
        getLocalPriority ( (int *) &(data->priority));
        *pDataSize = sizeof(VXDCOMEXTENT);
        }
    }
```

GUID_VXDCOM_EXTENT and **VXDCOMEXTENT** are defined in *installDir***target\h\dcomLib.h**. Windows programmers wishing to do priority propagation should #include this file. Once a channel hook is implemented, it must be registered with the Windows COM run-time using the **CoRegisterChannelHook( )** function.

9.8.3 Using Threadpools

A threadpool is a group of tasks owned by the VxWorks run-time libraries, dedicated to servicing DCOM requests for object method execution. Since a small number of tasks can handle a large number of requests, threadpools optimize performance and increase the scalability of the system.[4]

If all of the tasks in the pool are busy, new tasks can be dynamically added to the pool to handle the increased activity. These tasks are reclaimed by the system when the load drops again.

If all of the static tasks in the thread pool are busy and the maximum number of dynamic tasks has been allocated, then a queue can store the unserviced requests. As tasks become free, they will service the requests in this queue. If the queue becomes full, a warning message is returned to the calling task.

Since threadpools are part of the kernel, threadpool parameters are configurable.

9.9 Using OPC Interfaces

VxDCOM supports the OPC interfaces defined in the files listed below and identified by their corresponding categorical group name:

*installDir***target\h\opccomn.idl**	Common Interfaces
*nstallDir***target\h\opcda.idl**	Data Access
*installDir***target\h\opc_ae.idl**	Alarms and Events

[4] This is similar to a call center where a fixed number of operators service incoming calls. In fact, there are Erlang calculators that can optimize the ideal number of operators given the average call frequency and length.

Some of the OPC interface services use array and anonymous structure data types. In order to support these types as per the OPC specification, these files are shipped with some slight modifications in the form of conditional tags.

If you use OPC interfaces in your application, you may need to use non-automation data types that require additional editing in the **.idl** file. For details, see the *Tornado User's Guide: Building COM and DCOM Applications*.

The VxDCOM version of the OPC files import the *installDir***\target\h\vxidl.idl** file. This file defines interfaces required by an OPC application. For each of the interfaces that are required from the OPC protocol, a proxy/stub interface is required for VxDCOM to be able to remotely access an interface. In order to use this proxy/stub code, you must add the **DCOM_OPC** support component, as described in the *Tornado User's Guide: Building COM and DCOM Applications*.

9.10 Writing VxDCOM Servers and Client Applications

This section discusses programming issues and provides example code for writing VxDCOM servers and client applications.

9.10.1 Programming Issues

When writing VxDCOM applications, there are several programming issues of which you need to be aware. These are described below.

Using the VX_FP_TASK Option

The main program for all COM threads should be created using the **VX_FP_TASK** option.

NOTE: Because COM is single-threaded, any COM objects that are spawned by the main program will run within the main program thread.

Avoiding Virtual Base Classes

Each implementation of an interface must have its own copy of the vtable, including sections for the **IUnknown** method pointers. Using virtual inheritance alters this layout and interferes with the COM interface mapping.

Therefore, when defining your COM and DCOM classes, do not use virtual inheritance. For example, do *not* use this type of definition:

```
class CoServer : public virtual IFoo,
                 public virtual IBar
    {
    // class impl...
    };
```

Distinguishing the COM and DCOM APIs

VxDCOM supports both in-process and remote servers. When you write the servers, note that the main distinction between COM and DCOM components is the version of COM API routines used. The DCOM version of a COM routine is often appended with an **Ex**. In particular, you must use the **CoCreateInstanceEx( )** version of the COM **CoCreateInstance( )** routine for DCOM applications. For details, see the COM and DCOM libraries as part of the Microsoft COM documentation.

9.10.2 Writing a Server Program

This remainder of this section describes the server code for the **CoMathDemo**, focusing the methods that provide services to the client. These methods are the interface between client and server. The code used in this section is taken from the following files:

- the **CoMath.idl** interface definition file
- the **CoMathDemoImpl.h** header file
- the **CoMathDemoImpl.cpp** implementation file

For the implementation details, see the **CoMathDemo** source files, which are well commented.

Server Interfaces

The **CoMathDemo** server component implements two interfaces, **IMathDemo** and **IEvalDemo**, both of which are derived directly from **IUknown**.

The **IMathDemo** interface defines 19 common **math** methods listed below in the interface definition:

```
interface IMathDemo : IUnknown
    {
    HRESULT pi ([out,retval]double* value);
    HRESULT acos ([in]double x, [out,retval]double* value);
    HRESULT asin ([in]double x, [out,retval]double* value);
    HRESULT atan ([in]double x, [out,retval]double* value);
    HRESULT cos ([in]double x, [out,retval]double* value);
    HRESULT cosh ([in]double x, [out,retval]double* value);
    HRESULT exp ([in]double x, [out,retval]double* value);
    HRESULT fabs ([in]double x, [out,retval]double* value);
    HRESULT floor ([in]double x, [out,retval]double* value);
    HRESULT fmod ([in]double x, [in]double y, [out,retval]double* value);
    HRESULT log ([in]double x, [out,retval]double* value);
    HRESULT log10 ([in]double x, [out,retval]double* value);
    HRESULT pow ([in]double x, [in]double y, [out,retval]double* value);
    HRESULT sin ([in]double x, [out,retval]double* value);
    HRESULT sincos ([in]double x, [out]double* sinValue,
                    [out]double* cosValue);
    HRESULT sinh ([in]double x, [out,retval]double* value);
    HRESULT sqrt ([in]double x, [out,retval]double* value);
    HRESULT tan ([in]double x, [out,retval]double* value);
    HRESULT tanh ([in]double x, [out,retval]double* value);
    };
```

The **IEvalDemo** defines two methods, **eval( )** and **evalSubst( )**:

```
interface IEvalDemo : IUnknown
    {
    HRESULT eval ([in]BSTR str, [out,retval]double* value);
    HRESULT evalSubst ([in]BSTR str,
                       [in]double x,
                       [out,retval]double* value);
    };
```

The **eval( )** method takes a **BSTR**, which contains a algebraic expression, **str**, and returns the value as a double, **value**. Note that the variable that the **eval( )** method receives for evaluation is defined by the **[in]** parameter attribute and the value that is returned is defined by both the **[out]** and **[retval]** parameter attributes. The **[retval]** attribute allows this method to be used by languages that require a return value. For more information on parameter attributes, see the *Tornado User's Reference: COM Tools*.

This **evalSubst( )** method similarly takes a **BSTR** containing an algebraic expression and returns the value as a double. In addition, **evalSubst( )** will also substitute the variable **x** with a supplied value.

The implementation code indicates that both **eval( )** and **evalSubst( )** return an **HRESULT** of **S_OK** when successful or **E_FAIL** when an error occurs in decoding the expression.

Client Interaction

The **CoMathDemoClient** is created with arguments that include an expression to evaluate. The client first queries the server for the **IEvalDemo** interface and then for the **IMathDemo** interface by invoking the **QueryInterface( )** method of **IUknown**. The client code then calls the **eval( )** method of **IEvalDemo**, passing it the expression to evaluate, **str**, and a reference for the return value, **&result**. Then the client code is written specifically to call the **pi( )**, **sin( )**, and **cos( )** methods of the **IMathDemo** interface. (see *9.10.4 Querying the Server*, p.383).

9.10.3 Writing Client Code

Because DCOM is transparent across architectures, it minimizes the need for special code to be written in the client application that differentiates between in-process and remote procedure calls to an object. You can write code that uses the services of a COM interface, without concern for the network location of the object that implements that interface. The **MathDemo** program, described below, uses the same user client code. Only the **ifdef** statements generated by the VxDCOM wizard determine the whether the client is a COM or DCOM client and, for DCOM clients, whether it is for use on VxWorks or Windows NT.

This program demonstrates how to write a simple COM or DCOM client. The **MathDemo** program creates a COM object and uses that object to perform simple arithmetic calculations.

Determining the Client Type

The first section of the client code contain **#define** and **#include** directives that are mostly auto-generated by the VxDCOM wizard. The **#include** directives for the **alt*.*** files for **_WIN32** and the **comOjbLib.h** file were added manually because this program uses **CComBSTR**. Similarly, you would add any necessary header files for code that your program requires.

```
/* includes */

#ifdef _WIN32

#define _WIN32_WINNT 0x0400
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "CoMathDemo.w32.h"
#include <atlbase.h>
#include <atlimpl.cpp>
```

```
#else

#include "comLib.h"
#include "CoMathDemo.h"
#include "comObjLib.h"
#define mbstowcs comAsciiToWide

#endif

#include <stdio.h>
#include <iostream.h>
#include <math.h>

#ifdef _DCOM_CLIENT
#ifndef _WIN32
#include "dcomLib.h"
#endif
#endif
```

Creating and Initializing the Client

This section of the code creates the COM or DCOM object (component) by calling the appropriate COM or DCOM routines, **CoCreateInstance( )** or **CoCreateInstanceEx( )**. The DCOM client requires additional initialization code for security purposes. This creation and initialization code is auto-generated by the wizard.

```
#define MAX_X 79
#define MAX_Y 25

int CoMathDemoClient (const char* serverName, const char* expression)
    {
    HRESULT     hr = S_OK;
    double      result;
    int         x;
    int         y;
    IUnknown *  pItf;
    IEvalDemo * pEvalDemo;
    IMathDemo *  pMathDemo;
```

If the client is a COM client, this code is used:

```
#ifndef _DCOM_CLIENT
    // This section creates the COM object.
    hr = CoCreateInstance (CLSID_CoMathDemo,
                           0,
                           CLSCTX_INPROC_SERVER,
                           IID_IEvalDemo,
                           (void **)&pItf);
#else
```

If the client is DCOM client, this code is used. This section of code initializes DCOM for this thread and creates the DCOM object on the target:

```
OLECHAR wszServerName [128];

// Convert the server name to a wide string.
mbstowcs (wszServerName, serverName, strlen (serverName) + 1);

// Initialize DCOM for this thread.
hr = CoInitializeEx (0, COINIT_MULTITHREADED);
if (FAILED (hr))
    {
    cout << "Failed to initialize DCOM\n";
    return hr;
    }

// This initizlizes security to none.
hr = CoInitializeSecurity (0, -1, 0, 0,
                        RPC_C_AUTHN_LEVEL_NONE,
                        RPC_C_IMP_LEVEL_IDENTIFY,
                        0, EOAC_NONE, 0);
if (FAILED (hr))
    {
    cout << "Failed to initialize security\n";
    return hr;
    }
```

The code following creates an MQI structure, which is used to query the COM server object for the **IID_IMathDemo** interface. This is one of the two interfaces defined by the **CoMathDemo** CoClass that is instantiated as the DCOM server.

When writing a typical DCOM client program with multiple interfaces you include all your interface requests into the **MQI** and query them as one operation (thus saving bandwidth). However, for the purposes of this demo we want to keep the main body of the code the same; therefore, we only want the **IUnknown** for the DCOM object at this point so that we can treat it the same way as a COM object lower down:

```
MULTI_QI mqi [] = { {&IID_IEvalDemo, 0, S_OK} };
COSERVERINFO serverInfo = { 0, wszServerName, 0, 0 };

hr = CoCreateInstanceEx (CLSID_CoMathDemo,
                        0,
                        CLSCTX_REMOTE_SERVER,
                        &serverInfo,
                        1,
                           mqi);

if (SUCCEEDED (hr) && SUCCEEDED (mqi [0].hr))
    {
    cout << "Created CoMathDemo OK\n";
    pItf = mqi [0].pItf;
```

```
        }

    else

        {
        cout << "Failed to create CoMathDemo, HRESULT=" <<
            hex << cout.width (8) << mqi [0].hr << "\n";
        return E_FAIL;
        }

#endif
```

9.10.4 Querying the Server

Query the **IUnknown** interface of the COM object to get an interface pointer to the **IEvalDemo** interface:

```
if (FAILED (hr = pItf->QueryInterface (IID_IEvalDemo,
            (void**)&pEvalDemo)))
    {
    cout << "Failed to create IEvalDemo interface pointer,
            HRESULT=" << hex << cout.width (8) << hr << "\n";
    pItf->Release ();
    return hr;
    }
```

Query the **IUnknown** interface of the COM object to get an interface pointer to the **IMathDemo** interface:

```
if (FAILED (pItf->QueryInterface (IID_IMathDemo, void**)&pMathDemo)))
    {
    cout << "Failed to create IMathDemo interface pointer, HRESULT="
         << hex << cout.width (8) << hr << "\n";
    pEvalDemo->Release();
    pItf->Release ();
    return hr;
    }

pItf->Release ();
```

This code calls the **eval( )** method of the **IEvalDemo** interface, querying for it to evaluate the given expression, which is passed to the client program from the command line:[5]

```
cout << "Querying IEvalDemo interface\n";
CComBSTR str;
```

[5] The expression is passed in as an array of **char**, but it is converted to a **BSTR** for marshaling across to the COM server.

```
str = expression;

hr = pEvalDemo->eval(str, &result);
if (SUCCEEDED (hr))
    {
    cout << expression << "=" << result;
    }
    else
    {
    cout << "eval failed (" << hr << "," << result << ")\n";
    }
pEvalDemo->Release ();
```

This code queries the **IMathDemo** interface to draw the sine and cosine graphs. Note that it calls the **pi( )**, **sin( )**, and **cos( )** methods of that interface:

```
printf("Querying IMathDemo interface\n");

double sinResult;
double cosResult;
double pi;

hr = pMathDemo->pi(&pi);
if (FAILED (hr))
    return hr;

double step_x =  (pi * 2.0) / ((double)MAX_X);
double scale_y = ((double)MAX_Y) / 2.0;

for (y = MAX_Y; y >= 0; y--)
    {
    for (x = 0; x < MAX_X; x++)
        {
        hr = pMathDemo->sin((double)x * step_x , &sinResult);
        if (FAILED (hr))
            return hr;
        hr = pMathDemo->cos((double)x * step_x , &cosResult);
        if (FAILED (hr))
            return hr;
        if ((int)((double)((sinResult + 1.0) * scale_y)) == y)
            {
            putchar('*');
            }
        else if ((int)((double)((cosResult + 1.0) * scale_y)) == y)
            {
            putchar('+');
            }
        else
            {
            putchar(' ');
            }
        }
    putchar('\n');
    }
pMathDemo->Release ();
```

```
#ifdef _DCOM_CLIENT
    CoUninitialize ();
#endif
    return hr;
    }
```

9.10.5 Executing the Client Code

This section of code is for a DCOM C++ client running on a Windows NT operating system. This code was generated by the wizard except for the addition of one argument, **exp**, which bumps the number of arguments to check for up to 3:

```
#ifdef _WIN32
int main (int argc, char* argv [])
    {
    if (argc != 3)
    {
    puts ("usage: CoMathDemoClient <server> <exp>");
    exit (1);
    }

    return CoMathDemoClient (argv [1], argv[2]);
    }
```

This section of the code was added by the programmer for situations in which you do not want C++ name mangling:

```
#else
extern "C"
    {
    int ComTest (char * server, char * exp)
        {
        return CoMathDemoClient (server, exp);
        }
    }
#endif
```

9.11 Comparing VxDCOM and ATL Implementations.

This section is a reference summary of the noteworthy differences between the VxDCOM and Microsoft ATL implementations of COM interfaces. It also includes the VxDCOM implementation synopsis for **VARIANT**s.

9.11.1 CComObjectRoot

VxDCOM implements **CComObjectRoot** directly, whereas ATL implements it as a typedef of **CComObjectRootEx**.

CComObjectRoot always implements the aggregation pointer even if it is not used.

Constructor

Constructor for the class. Initializes the ref count to 0 and provides a storage for the aggregating outer interface, if required.

```
CComObjectRoot
    (
    IUnknown *  punk = 0        // aggregating outer
    )
```

InternalAddRef

Increments the ref count by one and returns the resultant ref count.

```
ULONG InternalAddRef ()
```

InternalRelease

Decrements the ref count by one and returns the resultant ref count.

```
ULONG InternalRelease ()
```

9.11.2 CComClassFactory

Derived from **IClassFactory** and **CComObjectRoot**.

Constructor

Constructor for class.

```
CComClassFactory ()
```

AddRef

Increments the ref count by one and returns the resultant ref count. This is handled by a call to **InternalAddRef( )** in **CComObjectRoot**.

```
ULONG STDMETHODCALLTYPE AddRef ()
```

Release

Decrements the ref count by one and returns the resultant ref count. This is handled by a call to **InternalRelease( )** in **CComObjectRoot**.

```
ULONG STDMETHODCALLTYPE Release ()
```

QueryInterface

Provides the **QueryInterface** mechanism to provide the **IID_IUnknown** and **IID_IClassFactory** interface to **IClassFactory**.

```
HRESULT STDMETHODCALLTYPE QueryInterface
    (
    REFIID riid, // The GUID of the interface being requested
    void ** ppv // A pointer to the interface riid
    )
```

CreateInstance

Creates a new instance of the requested object; calls **CComObjectRoot::CreateInstance**. Therefore, unlike ATL, the object created for WOTL is initialized.

```
HRESULT STDMETHODCALLTYPE CreateInstance
    (
    IUnknown * pUnkOuter, // aggregated outer
    REFIID riid, // The GUID of the interface being created
    void ** ppv // A pointer to the interface riid
    )
```

LockServer

This is a stub function, which always returns **S_OK** and is provided for compatibility only.

```
HRESULT STDMETHODCALLTYPE LockServer
    (
    BOOL Lock
    )
```

9.11.3 CComCoClass

VxDCOM does not implement the macros **DECLARE_CLASSFACTORY** or **DECLARE_AGGREGATABLE**. The functionality is implicitly built into the class.

GetObjectCLSID

Returns the **CLSID** of the object.

```
static const CLSID& GetObjectCLSID( )
```

9.11.4 CComObject

Derived from the given interface class.

CreateInstance

There are two versions of **CreateInstance**. For details about when each is used and how it is invoked, see the Microsoft COM documentation.

CreateInstance as defined below creates a single instance with no aggregation and no specific COM interface.

```
static HRESULT CreateInstance
    (
    CComObject** pp // object that is created
    )
```

The version of **CreateInstance** below creates an instance of the class and searches for the requested interface on it.

```
static HRESULT CreateInstance
    (
    IUnknown* punkOuter, // aggretable interface
    REFIID riid, // GUID of the interface
    void** ppv // resultant object
    )
```

AddRef

Does an **AddRef** on either the **punkOuter** or the **CComObjectRoot** depending on the type of the object.

```
ULONG STDMETHODCALLTYPE AddRef ()
```

Release

Does a **Release** on either the **punkOuter** or the **CComObjectRoot::AddRef** depending on the type of object.

```
ULONG STDMETHODCALLTYPE Release ()
```

QueryInterface

Queries either the **punkOuter** or the object for an interface.

```
HRESULT STDMETHODCALLTYPE QueryInterface
    (
    REFIID riid, // GUID to query for
    void ** ppv // resultant interface
    )
```

9.11.5 CComPtr

Template class that takes a COM interface specifying the type of pointer to be stored.

Constructors

Supported constructors.

```
CComPtr ()
CComPtr (Itf* p)
CComPtr (const CComPtr& sp)
```

Release

Release an instance of the object.

```
void Release ()
```

Operators

Supported operators.

```
operator Itf* () const
Itf** operator& ()
Itf* operator-> ()
const Itf* operator-> () const
Itf* operator= (Itf* p)
Itf* operator= (const CComPtr& sp)
bool operator! () const
```

Attach

Attach an object to the pointer without incrementing its ref count.

```
void Attach
    (
    Itf * p2 // object to attach to pointer
    )
```

Detach

Detach an object from the pointer without decrementing its ref count.

```
Itf *Detach()
```

CopyTo

Copy an object to the pointer and increment its ref count.

```
HRESULT CopyTo
    (
    Itf ** ppT // object to copy to pointer
    )
```

9.11.6 CComBSTR

Class wrapper for the **BSTR** type. **CComBSTR** provides methods for the safe creation, assignment, conversion and destruction of a **BSTR**.

Constructors

Supported constructors.

```
CComBSTR ()
explicit CComBSTR (int nSize, LPCOLESTR sz = 0)
explicit CComBSTR (LPCOLESTR psz)
explicit CComBSTR (const CComBSTR& src)
```

Operators

Supported operators.

```
CComBSTR& operator= (const CComBSTR& cbs)
CComBSTR& operator= (LPCOLESTR pSrc)
operator BSTR () const
BSTR * operator& ()
bool operator! () const
CComBSTR& operator+= (const CComBSTR& cbs)
```

Length

Get the length of the **BSTR**.

```
unsigned int Length () const
```

Copy

Make a copy of the BSTR within the wrapper class and return it.

```
BSTR Copy() const
```

Append

Append to the **BSTR**.

```
void Append (const CComBSTR& cbs)
void Append  (LPCOLESTR lpsz)
void AppendBSTR (BSTR bs)
void Append (LPCOLESTR lpsz, int nLen)
```

Empty

Delete any existing **BSTR** from the wrapper class.

```
void Empty ()
```

Attach

Attach a **BSTR** to the wrapper class.

```
void Attach (BSTR src)
```

Detach

Detach the **BSTR** from the wrapper class and return it.

```
BSTR Detach ()
```

9.11.7 VxComBSTR

The **comObjLibExt** file provides VxWorks specific extensions to the existing ATL-like classes defined in **comObjLib.h**. **VxComBSTR** is derived from **CComBSTR** and extends it.

Constructors

Constructors for this class are derived from the **CComBSTR** constructors.

```
VxComBSTR ()
explicit VxComBSTR (int nSize, LPCOLESTR sz = 0)
explicit VxComBSTR (const char * pstr)
explicit VxComBSTR (LPCOLESTR psz)
explicit VxComBSTR (const CComBSTR& src)
explicit VxComBSTR (DWORD src)
explicit VxComBSTR (DOUBLE src)
```

Operators

Supported operators are described below, followed by the declaration.

Convert the **BSTR** into an array of **char** and return a pointer to a temporary copy. This copy is guaranteed to be valid until another call is made on the **VxComBSTR** object. It can be used in place of the **OLE2T** macro:

```
operator char * ()
```

Return the decimal numeric value stored in the **BSTR** as a **DWORD** value. This method follows the same string rules as the standard library function **atoi**:

```
operator DWORD ()
```

Convert a **DWORD** value into its decimal string representation and store it as a **BSTR**:

```
VxComBSTR& operator = (const DWORD& src)
```

Convert a **DOUBLE** value into it's decimal string representation and stores it as a **BSTR**:

```
VxComBSTR& operator = (const DOUBLE& src)
```

Convert an array of char into a **BSTR** format. It can be used instead of **T2OLE**:

```
VxComBSTR& operator = (const char * src)
```

Return **TRUE** if the given **VxComBSTR** value is equal to the stored **BSTR**, **FALSE** otherwise:

```
bool const operator == (const VxComBSTR& src)
```

Return **TRUE** if the given **VxComBSTR** value is not equal to the stored **BSTR**, **FALSE** otherwise:

```
bool const operator != (const VxComBSTR& src)
```

SetHex

Convert a **DWORD** value into its hexadecimal string representation and stored it as a **BSTR**.

```
void SetHex (const DWORD src)
```

9.11.8 CComVariant

Derived from **tagVARIANT**.

Constructors

Supported Constructors.

```
CComVariant()
CComVariant(const VARIANT& varSrc)
CComVariant(const CComVariant& varSrc)
CComVariant(BSTR bstrSrc)
CComVariant(LPCOLESTR lpszSrc)
CComVariant(bool bSrc)
CComVariant(int nSrc)
CComVariant(BYTE nSrc)
CComVariant(short nSrc)
CComVariant(long nSrc, VARTYPE vtSrc = VT_I4)
CComVariant(float fltSrc)
CComVariant(double dblSrc)
CComVariant(CY cySrc)
CComVariant(IUnknown* pSrc)
```

Operators

Supported operators.

```
CComVariant& operator=(const CComVariant& varSrc)
CComVariant& operator=(const VARIANT& varSrc)
CComVariant& operator=(BSTR bstrSrc)
CComVariant& operator=(LPCOLESTR lpszSrc)
CComVariant& operator=(bool bSrc)
CComVariant& operator=(int nSrc)
CComVariant& operator=(BYTE nSrc)
CComVariant& operator=(short nSrc)
CComVariant& operator=(long nSrc)
CComVariant& operator=(float fltSrc)
CComVariant& operator=(double dblSrc)
CComVariant& operator=(CY cySrc)
CComVariant& operator=(IUnknown* pSrc)
bool operator==(const VARIANT& varSrc)
bool operator!=(const VARIANT& varSrc)
```

Clear

Clear the **VARIANT** within the wrapper class.

```
HRESULT Clear()
```

Copy

Copy the given **VARIANT** into the wrapper class.

```
HRESULT Copy(const VARIANT* pSrc)
```

Attach

Attach the given **VARIANT** to the class without copying it.

```
HRESULT Attach(VARIANT* pSrc)
```

Detach

Detach the **VARIANT** from the class and return it.

```
HRESULT Detach(VARIANT* pDest)
```

ChangeType

Change the type of the **VARIANT** to **vtNew**. The types supported by this wrapper function are the same as those of the **comLib** function **VariantChangeType**.

```
HRESULT ChangeType
    (
    VARTYPE vtNew,
    const VARIANT* pSrc = NULL
    )
```

10

Distributed Message Queues

Optional Component VxFusion

10.1 Introduction

VxFusion is a lightweight, media-independent mechanism, based on VxWorks message queues, for developing distributed applications.

There are several options for distributed multiprocessing in VxWorks. The Wind River optional product VxMP allows objects to be shared, but only across shared memory. TCP/IP can be used to communicate across networks, but it is low level and not intended for real-time use. Various high-level communication mechanisms that are standard for distributed computing can be used, but they have high overheads in terms of memory usage and computation time that are not always acceptable for real-time systems. Numerous proprietary methods also have been developed, but more and more often they encounter maintenance, porting, and enhancement issues. VxFusion, however, is a standard VxWorks component that:

- Provides a lightweight distribution mechanism based upon VxWorks message queues.
- Provides media independence, allowing distributed systems to effectively exchange data over any transport, eliminating custom requirements for communications hardware.
- Provides a safeguard against a single point failure resulting from one-to-many master slave dependencies. This is done by replicating a database of known objects on every node in the multi-node system.

- Supports unicast and multicast posting of messages to objects in the system.
- Exhibits location transparency; that is, objects can be moved seamlessly within the system without rewriting application code. Specifically, posting messages to objects occurs without regard to their location in a multi-node system.

VxFusion is similar to the VxMP shared memory objects option. VxMP adds to the basic VxWorks message queue functionality support for sharing message queues, semaphores, and memory allocation over shared memory. VxFusion adds to VxWorks support for sharing message queues over any transport, as well as multicasting to message queue groups. Unlike the VxMP shared memory option, VxFusion does not support distributed semaphores or distributed memory allocation.

⚠ **CAUTION:** When a distributed queue is created in one host, this information is broadcast to the other host through the name database. If the host where the queue was created crashes, there is no easy way for the other host to find out this information. Thus, the other host might be pending on a receive forever. It is up to the user to provide ways to detect remote nodes crashes and update the database accordingly.

10.2 Configuring VxWorks with VxFusion

Configure VxWorks with the **INCLUDE_VXFUSION** component to provide basic VxFusion functionality. To configure VxFusion show routines, also include the following components in the kernel domain:

INCLUDE_VXFUSION_DIST_MSG_Q_SHOW
INCLUDE_VXFUSION_GRP_MSG_Q_SHOW
INCLUDE_VXFUSION_DIST_NAME_DB_SHOW
INCLUDE_VXFUSION_IF_SHOW

NOTE: VxFusion cannot be configured with targets that do not support Ethernet broadcasting, such as the VxSim target simulator.

10.3 Using VxFusion

VxFusion adds two types of distributed objects to the standard VxWorks message queue functionality:

- distributed message queues
- group message queues

Distributed message queues can be shared over any communications medium. Group message queues are virtual message queues that receive a message, then send it out to all message queue members of the group. VxFusion implements a set of routines that offer fine-tuned control over the timing of distributed message queue operations. However, many of the API calls used to manipulate standard message queues also work with distributed objects, making the porting of your message-queue-based application to VxFusion easy.

VxFusion also provides a distributed name database. The distributed name database is used to share data between applications. The API for the distributed name database is similar to the API for the shared name database of VxMP. See the entry for **distNameLib** in the *VxWorks API Reference*.

This section discusses system architecture and initialization, configuring VxFusion, working with the various components, and writing an adapter.

10.3.1 VxFusion System Architecture

A typical VxFusion system consists of two or more nodes connected by a communications path, as shown in Figure 10-1. The communications path is known as the *transport*, which can be communications hardware, like an Ethernet interface card or a bus, or it can be a software interface to communications hardware, like a driver or protocol stack. Most often the transport is a software interface; rarely is it useful or essential to communicate directly with the communications hardware.

There are many possible transports and, thus, many possible different APIs, addressing schemes, and so on; therefore, VxFusion requires a piece of software, called an *adapter*, to be placed between itself and the transport. The adapter provides a uniform interface to VxFusion, regardless of the underlying transport. In this way, an adapter is similar to a driver. And, as with drivers, a new adapter must be written for each type of transport VxFusion supports. The VxFusion component supplies a sample UDP adapter to be used as is or as a guide for writing new adapters (see *10.3.7 Working with Adapters*, p.417).

Figure 10-1 **Example VxFusion System**

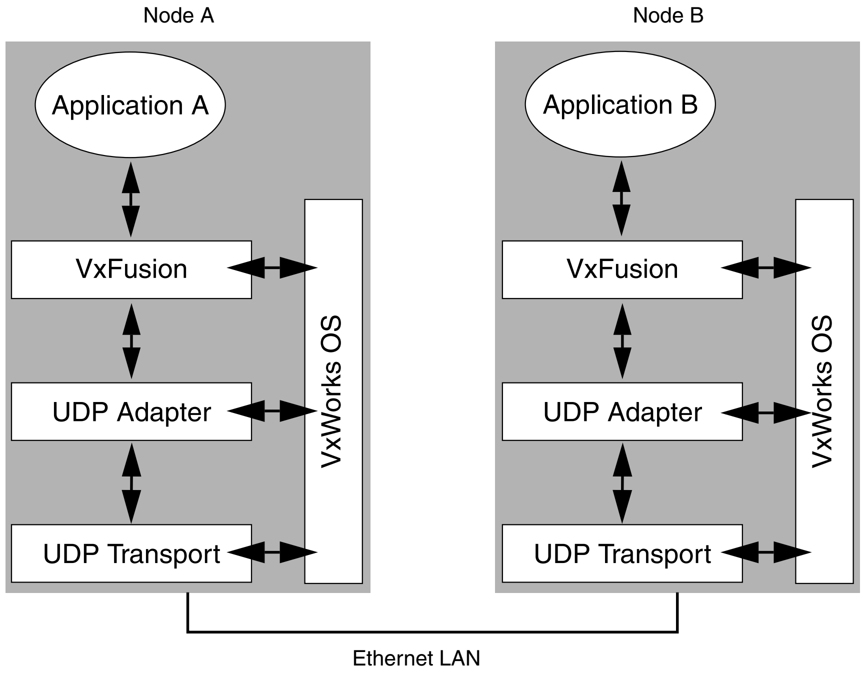


Figure 10-1 illustrates an example two-node VxFusion system. VxFusion has been installed on each of the two nodes that are connected to the same subnet of an Ethernet LAN. Because the two nodes are connected by Ethernet, TCP, UDP, IP, and raw Ethernet are all possible transports for communications.

In Figure 10-1, the UDP protocol serves as the transport and the supplied UDP adapter as the adapter.

Services and Databases

VxFusion is actually made up of a number of services and databases, as shown in Figure 10-2. The VxFusion databases and services are listed in Table 10-1 and Table 10-2, respectively (the term *telegram* is used in these tables; see *10.6 Telegrams and Messages*, p. 421). Each service runs as a separate task; you see these tasks identified whenever you list tasks on a node with VxFusion installed and running.Note that you need not be aware of these services and databases to use VxFusion.

Figure 10-2 **VxFusion Components**

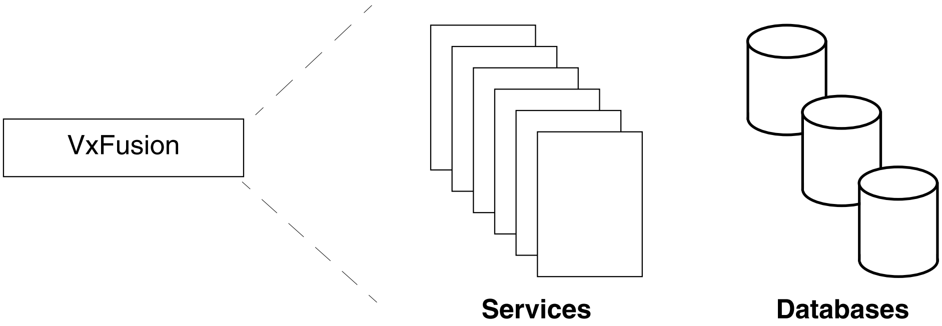


Table 10-1 **VxFusion Databases**

Database	Description
Distributed Name Database	The distributed name database consists of name-value- type entries (see *10.3.4 Working with the Distributed Name Database*, p.404). It has copies on every node in the system, making entries available to tasks on any node.
Distributed Group Database	The distributed group database maintains the list of distributed message queue groups and their locally added members.
Distributed Node Database	The distributed node database maintains the list of all other nodes in the system along with their status.

Table 10-2 **VxFusion Services**

Service	Description
Distributed Message Queue Service	Handles distributed message queue telegrams from remote nodes.
Group Message Queue Service	Handles group message queue telegrams from remote nodes.
Distributed Name Database Service	Handles distributed name database telegrams from remote nodes.
Distributed Group Database Service	Handles distributed message queue group database telegrams from remote nodes.
Incorporation Service	Handles incorporation messages from remote nodes. Incorporation messages are used to signal and acknowledge the start and end of database updates.
Group Agreement Protocol (GAP) Service	Handles GAP messages. GAP messages are sent between nodes to choose a unique ID for groups.

Libraries

VxFusion provides the following software libraries:

a distributed name database
(**distNameLib**, **distNameShow**)

distributed message queues
(**msgQDistLib**, **msgQDistShow**)

group message queues
(**msgQDistGrpLib**, **msgQDistGrpShow**)

telegram buffers
(**distTBufLib**)

distributed objects statistics
(**distStatLib**)

VxFusion adapter interface
(**distIfLib**, **distIfShow**)

VxFusion network layer
(**distNetLib**)

10.3.2 VxFusion Initialization

When you boot a VxWorks image that has VxFusion enabled, the **usrVxFusionInit( )** routine found in **usrVxFusion.c** calls the VxFusion initialization routine, **distInit( )**, to start VxFusion. The **distInit( )** routine initializes VxFusion on the current node. VxFusion must be installed on each node in a system for applications on those nodes to be able to communicate. Parameters of **distInit( )** are set to their default values. The **distInit( )** routine performs the following basic operations:

– Initializes the local services and databases.

– Identifies the other VxFusion nodes in the system and determines their status.

– If there are other nodes in the system, updates the local databases using data from one of the other nodes.

Because **distInit( )** is called automatically when a target boots, if your VxWorks image has VxFusion included, you should not call this routine directly from the user application.

10.3.3 Configuring VxFusion

You can configure various aspects of VxFusion—initialization, run-time activities, and the adapter interface—using one of the following methods:

- **usrVxFusionInit( )** to modify features set at initialization.
- **distCtl( )** to control run-time behavior.
- **DIST_IF** structures to tune adapter interfaces.

Each method for customizing is described further in this section.

Customizing with usrVxFusionInit( )

This routine invokes the VxFusion initialization routine **distInit( )**. It is customizable and can be used to modify **distInit( )** configuration parameters that control initialization of your VxFusion environment. Table 10-3 describes the numerous configurable arguments. For more information on **distInit( )**, see the entry in the *VxWorks API Reference*.

Table 10-3 **Configuration Parameters Modified Within usrVxFusionInit( )**

Parameter/Definition	Default Value	Description
myNodeId node ID	IP address of booting interface	Specify a unique ID for a node. Each node in a VxFusion system must have a unique node ID. By default, the **usrVxFusionInit( )** code uses the IP address of the booting interface as the node ID. The **usrVxFusionInit( )** routine provides this as the first argument to the **distInit( )** routine.
ifInitRtn adapter-specific initialization routine	**distUdpInit( )**	Specify the initialization routine of the interface adapter to be used. By default, the **usrVxFusionInit( )** routine specifies the initialization routine of the UDP adapter.
pIfInitConf adapter-specific configuration structure	booting interface	Provide any additional adapter-specific configuration information to the adapter. For example, the **usrVxFusionInit( )** routine provides the UDP adapter with the interface over which the node was booted.
maxTBufsLog2 maximum number of TBufs	9 (512 log 2)	Specify the maximum number of telegram buffers to create. The **usrVxFusionInit( )** routine must provide this parameter in log 2 form, that is, if the maximum is 512, the parameter must be 9.

Table 10-3 **Configuration Parameters Modified Within usrVxFusionInit( )** *(Continued)*

Parameter/Definition	Default Value	Description
maxNodesLog2 maximum number of nodes in the distributed node database	5 (32 log 2)	Specify the maximum number of nodes in the distributed node database. The **usrVxFusionInit( )** routine must provide this parameter in log 2 form.
maxQueuesLog2 *maximum number of* queues on node	7 (128 log 2)	Specify the maximum number of distributed message queues that can be created on a single node. The **usrVxFusionInit( )** routine must provide this parameter in log 2 form.
maxGroupsLog2 maximum number of groups in the distributed group database	6 (64 log 2)	Specify the maximum number of group message queues that can be created in the distributed group database. The **usrVxFusionInit( )** routine must provide this parameter in log 2 form.
maxNamesLog2 *maximum number of entries in the distributed* name database	8 (256 log 2)	Specify the maximum number of entries that can be stored in the distributed name database. The **usrVxFusionInit( )** routine must provide this parameter in log 2 form.
waitNTicks maximum number of clock ticks to wait	240*	Specify the number of clock ticks to wait for responses from other nodes at startup.

* 4***sysClkRateGet( )** is typically 240, but not always. The defaults are defined in **vxfusion/distLib.h** and can be changed by the user, if desired.

NOTE: If **distInit( )** fails, it returns ERROR, and VxFusion is not started on the host.

Customizing with distCtl( )

This routine performs a distributed objects control function. Use **distCtl( )** to configure run-time-related parameters and hooks listed in Table 10-4. For more information about available control functions using **distCtl( )**, see the entry in the *VxWorks API Reference*.

Table 10-4 **Configuration Parameters Modified Using distCtl( )**

Parameter or Hook	Default Value	Description
DIST_CTL_LOG_HOOK	NULL	Set a routine to be called each time a log message is produced. If no log hook is set, the log output is printed to standard output.
DIST_CTL_PANIC_HOOK	NULL	Set a routine to be called when the system panics due to an unrecoverable error. If no panic hook is set, the output is printed to standard output.
DIST_CTL_RETRY_TIMEOUT	200ms	Set the initial send retry timeout. Although the default timeout is shown in milliseconds, the retry timeout is actually set using clock ticks. Timeout values are rounded down to the nearest 200 ms.
DIST_CTL_MAX_RETRIES	5	Set the limit for the number of retries when sending fails.
DIST_CTL_NACK_SUPPORT	TRUE	Enable or disable the sending of negative acknowledgments (NACK).
DIST_CTL_PGGYBAK_UNICST_SUPPORT	FALSE	Enable or disable unicast piggy-backing.
DIST_CTL_PGGYBAK_BRDCST_SUPPORT	FALSE	Enable or disable broadcast piggy-backing.
DIST_CTL_OPERATIONAL_HOOK	NULL	Add to a list of routines to be called each time a node shifts to the operational state. Up to 8 routines can be added.
DIST_CTL_CRASHED_HOOK	NULL	Add to a list of routines to be called each time a node shifts to the crashed state. The list can hold a maximum of 8 routines; however, one space is used by VxFusion, leaving space for only 7 user-specified routines to be added.
DIST_CTL_SERVICE_HOOK	NULL	Set a routine to be called each time a service fails on a node, for a service invoked by a remote node.
DIST_CTL_SERVICE_CONF	service-specific	Set the task priority and network priority of the service.

NOTE: DIST_CTL_CRASHED_HOOK should always be used with **distCtl( )**, as it is the only way that VxFusion can provide notification that a node has crashed.

Customizing with DIST_IF Structures

Table 10-5 lists configurable fields of the **DIST_IF** structure, which is used to pass data about the adapter to VxFusion. It is the **DIST_IF** structure that gives VxFusion transport independence. For more information on the **DIST_IF** structure, see *10.7.2 Writing an Initialization Routine*, p. 425.

Table 10-5 **Configuration Parameters Modified Within Interface Adapters**

DIST_IF Field/ Definition	**Default Value**	**Description**
distIfName interface adapter name	adapter-specific UDP adapter: "UDP adapter"	Specifies the name of the interface adapter.
distIfMTU MTU size	adapter-specific UDP adapter: 1500 bytes	Specifies the MTU size of the interface adapter protocol.
distIfHdrSize network header size	adapter-specific UDP adapter: size of **NET_HDR**	Specifies the network header size.
distIfBroadcastAddr broadcast address	adapter-specific UDP adapter: IP broadcast address for subnet	Specifies the broadcast address for the interface and transport that broadcasts are to be sent on.
distIfRngBufSz ring buffer size	adapter-specific UDP adapter: 256	Specifies the number of elements to use in the sliding window protocol.
distIfMaxFrags maximum number of fragments	adapter-specific UDP adapter: 10	Specifies the maximum number of fragments into which a message can be broken.

10.3.4 Working with the Distributed Name Database

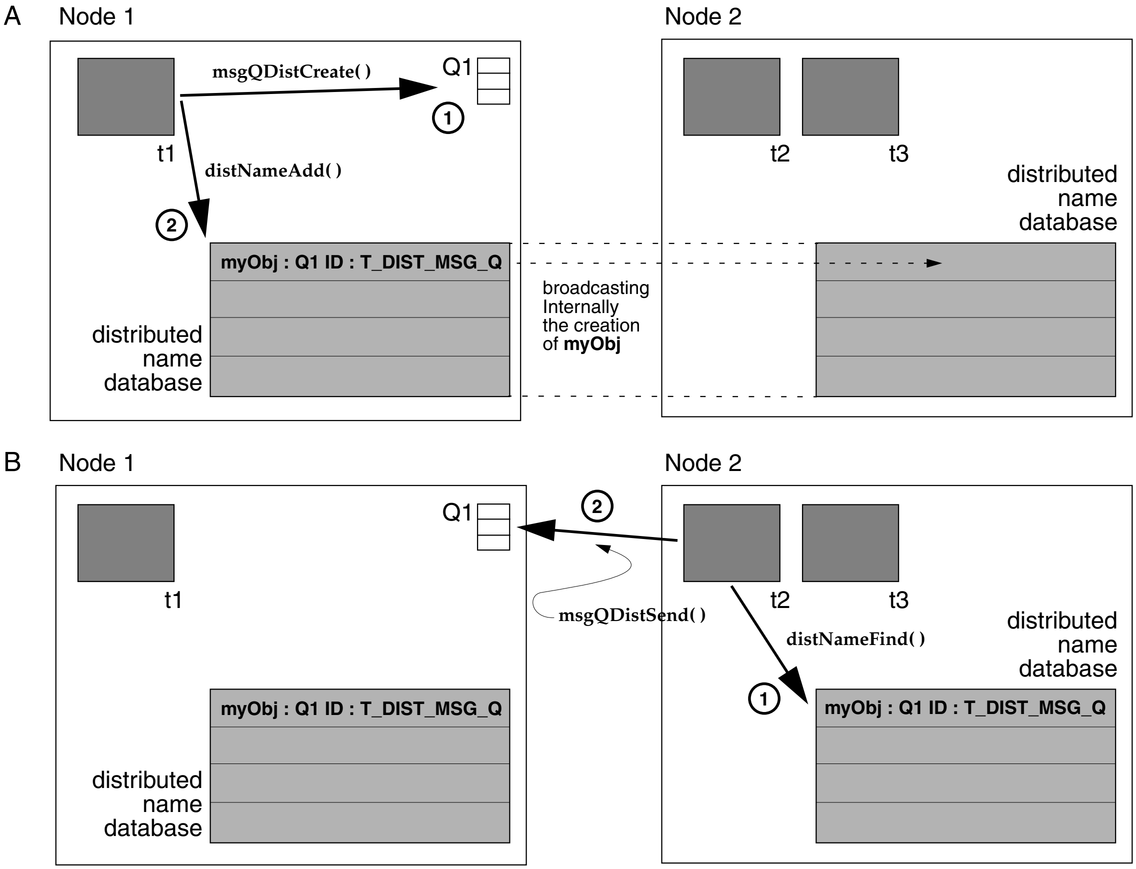
The distributed name database allows the association of any value to any name, such as a distributed message queue's ID with a unique name. The distributed name database provides name-to-value-and-type and value-and-type-to-name translation, allowing entries in the database to be accessed either by name or value and type.

Each node in a system has a copy of the distributed name database. Any modifications made to a local copy of the database are immediately sent to all other copies on all other nodes in the system.

Typically, the task that wants to share a value adds a name-value-type entry into the distributed name database. When adding the entry to the database, the task associates the value with a unique, specified name. Tasks on different nodes use this name to get the associated value.

Consider the example in Figure 10-3, which shows how two tasks on different nodes share a common distributed message queue ID.

Figure 10-3 **Using the Distributed Name Database**



Task **t1** on Node 1 creates a message queue. The distributed message queue ID is returned by the creation routine. Task **t1** adds the ID and its associated name, **myObj**, to the distributed name database. This database entry is then *broadcast* to all nodes in the system. For task **t2** on Node 2 to send a message to this distributed message queue, it first finds the ID by looking up the name **myObj** in Node 2's local copy of the distributed name database.

However, if node 2 does not receive the broadcast—because, for example, the network is down—the information on the two nodes does not match, and Node 2 is not aware of Q1. (This is an example of a case in which **DIST_CTL_CRASHED_HOOK** could be used with **distCtl( )** for notification; see Table 10-4.)

Table 10-6 lists the distributed name database service routines. The distributed name database may contain floating point values because they invoke **printf( )** to print them. Any task calling **distNameShow( )** should set the **VX_FP_TASK** task option set.

Additional information about adding a name to the distributed names database and about related show routines is provided in this section. For detailed information about all of these routines, see the entries in the *VxWorks API Reference*.

Table 10-6 **Distributed Name Database Service Routines**

Routine	Functionality
distNameAdd( )	Adds a name to the distributed name database.
distNameFind( )	Finds a distributed object by name.
distNameFindByValueAndType( )	Finds a distributed object by value and type.
distNameRemove( )	Removes an object from the distributed name database.
distNameShow( )	Displays the entire contents of the distributed name database to the standard output device.
distNameFilterShow( )	Displays all entries in the database of a specified type.

Adding a Name to the Distributed Name Database

Use **distNameAdd( )** to add a name-value-type entry into the distributed name database. The type can be user defined or pre-defined in **distNameLib.h.**

NOTE: The distributed name database service routines automatically convert to or from network-byte order for the pre-defined types only. Do not call **htnol( )** or **ntohl( )** explicitly for values of pre-defined types from the distributed name database.

Table 10-7 lists the pre-defined types.

Table 10-7 **Distributed Name Database Types**

Constant	Decimal Value	Purpose
T_DIST_MSG_Q	0	distributed message queue identifier (also group ID)
T_DIST_NODE	16	node identifier
T_DIST_UINT8	64	8-bit unsigned integer
T_DIST_UINT16	65	16-bit unsigned integer
T_DIST_UINT32	66	32-bit unsigned integer
T_DIST_UINT64	67	64-bit unsigned integer
T_DIST_FLOAT	68	single-precision floating-point number (32-bit)
T_DIST_DOUBLE	69	double-precision floating-point number (64-bit)
user-defined types	4096 and above	user-defined types

The value bound to a particular name can be updated by simply calling **distNameAdd( )** another time with a new value.

Displaying Distributed Name Database Information

There are two routines for displaying data from the distributed name database: **distNameShow( )** and **distNameFilterShow( )**.

⚠ **CAUTION:** The distributed name database provided by vxFusion may contain floating point values. The **distNameShow( )** routine invokes **printf( )** to print them. Any task calling **distNameShow( )** should set the **VX_FP_TASK** task option. The target shell has this option set.

The **distNameShow( )** routine displays the entire contents of the distributed name database to the standard output device. The following demonstrates use of **distNameShow( )**; the output is sent to the standard output device:

```
[VxKernel]-> distNameShow()

        NAME                 TYPE                VALUE
-------------------- -------------- -------------------------
nile                   T_DIST_NODE 0x930b2617 (2466981399)
columbia               T_DIST_NODE 0x930b2616 (2466981398)
dmq-01                 T_DIST_MSG_Q  0x3ff9fb
```

```
dmq-02                  T_DIST_MSG_Q 0x3ff98b
dmq-03                  T_DIST_MSG_Q 0x3ff94b
dmq-04                  T_DIST_MSG_Q 0x3ff8db
dmq-05                  T_DIST_MSG_Q 0x3ff89b
gData                   4096 0x48 0x65 0x6c 0x6c 0x6f 0x00
gCount                  T_DIST_UINT32 0x2d (45)
grp1                    T_DIST_MSG_Q 0x3ff9bb
grp2                    T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

The **distNameFilterShow( )** routine displays the contents of the distributed name database filtered by type. That is, it displays only the entries in the database that match the specified type. The following output illustrates use of **distNameFilterShow( )** to display only the message queue IDs:

```
[VxKernel]-> distNameFilterShow(0)

        NAME              TYPE               VALUE
-------------------- -------------- -------------------------
dmq-01                  T_DIST_MSG_Q 0x3ff9fb
dmq-02                  T_DIST_MSG_Q 0x3ff98b
dmq-03                  T_DIST_MSG_Q 0x3ff94b
dmq-04                  T_DIST_MSG_Q 0x3ff8db
dmq-05                  T_DIST_MSG_Q 0x3ff89b
grp1                    T_DIST_MSG_Q 0x3ff9bb
grp2                    T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

10.3.5 Working with Distributed Message Queues

Distributed message queues are message queues that can be operated on transparently by both local and remote tasks. Table 10-8 lists the routines used to control distributed message queues.

Table 10-8 **Distributed Message Queue Routines**

Routines	Functionality
msgQDistCreate( )	Creates a distributed message queue.
msgQDistSend( )	Sends a message to a distributed message queue.
msgQDistReceive( )	Receives a message from a distributed message queue.
msgQDistNumMsgs( )	Gets the number of messages queued to a distributed message queue.

A distributed message queue must be created using the **msgQDistCreate( )** routine. It physically resides on the node that instigated the create call.

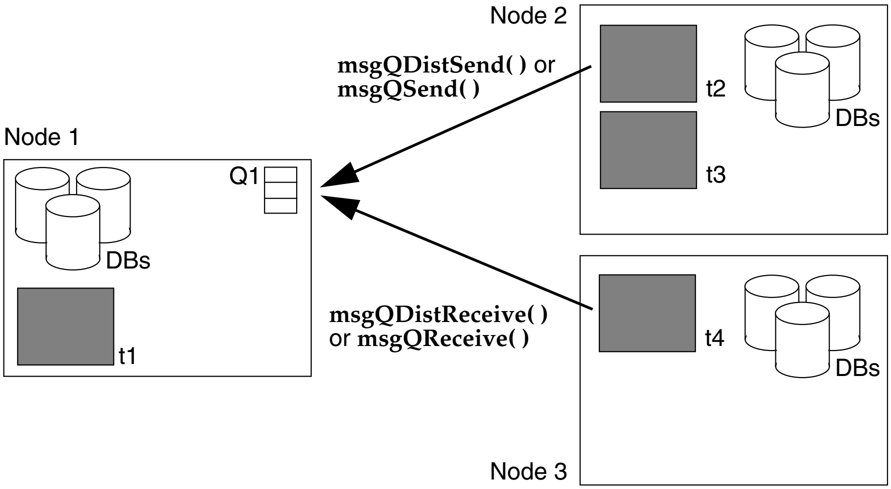
Once created, a distributed message queue can be operated on by the standard message queue routines provided by **msgQLib**, which are **msgQSend( )**, **msgQReceive( )**, **msgQNumMsgs( )**, **msgQDelete( )**, and **msgQShow( )**. This newly created distributed message queue is not available to remote nodes until the local node uses **distNameAdd( )** to add the ID to the distributed name database.

When using the standard message queue routines on a distributed message queue, the *timeout* argument specifies the amount of time to wait at the remote message queue only—there is no mechanism for indicating transmission time between nodes. When using the send, receive, and number-of-messages routines designed specifically for distributed message queues (**msgQDistSend( )**, **msgQDistReceive( )**, and **msgQDistNumMsgs( )**), you can take advantage of an additional timeout parameter, *overallTimeout*, that accounts for the transmission time as well.

NOTE: For this release, you cannot delete a distributed message queue. There is no **msgQDistDelete( )** routine, and a call to **msgQDelete( )** with a distributed message queue ID always returns an error.

Figure 10-4 illustrates send and receive operations.

Figure 10-4 **Sending to and Receiving from a Remote Distributed Message Queue**



However, before send and receive operations can occur, a task on Node 1 must have created a distributed message queue. The remote message queue **Q1** has been previously created by task **t1** on Node 1 with a call to **msgQDistCreate( )**. **Q1** was then added to the distributed name database with a call to **distNameAdd( )**. Tasks **t2** and **t4** have also previously obtained the remote message queue ID for **Q1** from the distributed name database using the **distNameFind( )** routine. With this data, task **t2** on Node 2 can send a message to **Q1**, using either the standard **msgQSend( )** routine or the VxFusion-specific **msgQDistSend( )** routine. Similarly, task **t4** on Node 3 can receive a message from **Q1** using the standard **msgQReceive( )** routine or the VxFusion-specific **msgQDistReceive( )** routine.

For detailed information about distributed message queue routines, see the entries in the *VxWorks API Reference*.

Sending Limitations

Local sending—that is, send actions on a single node—occurs in the same manner for distributed message queues as for standard message queues, and, therefore, is not discussed in greater detail in this manual. However, sending messages to remote message queues using the **msgQDistSend( )** routine can have different outcomes, depending on the value specified for the timeout arguments. Figure 10-5 presents three examples of messages being sent to a remote node. There are two threads of execution: the local node waits for the status of the send action, and the remote node waits to place the data in the distributed message queue. Both threads are controlled with timeouts, if **msgQDistSend( )** is used.

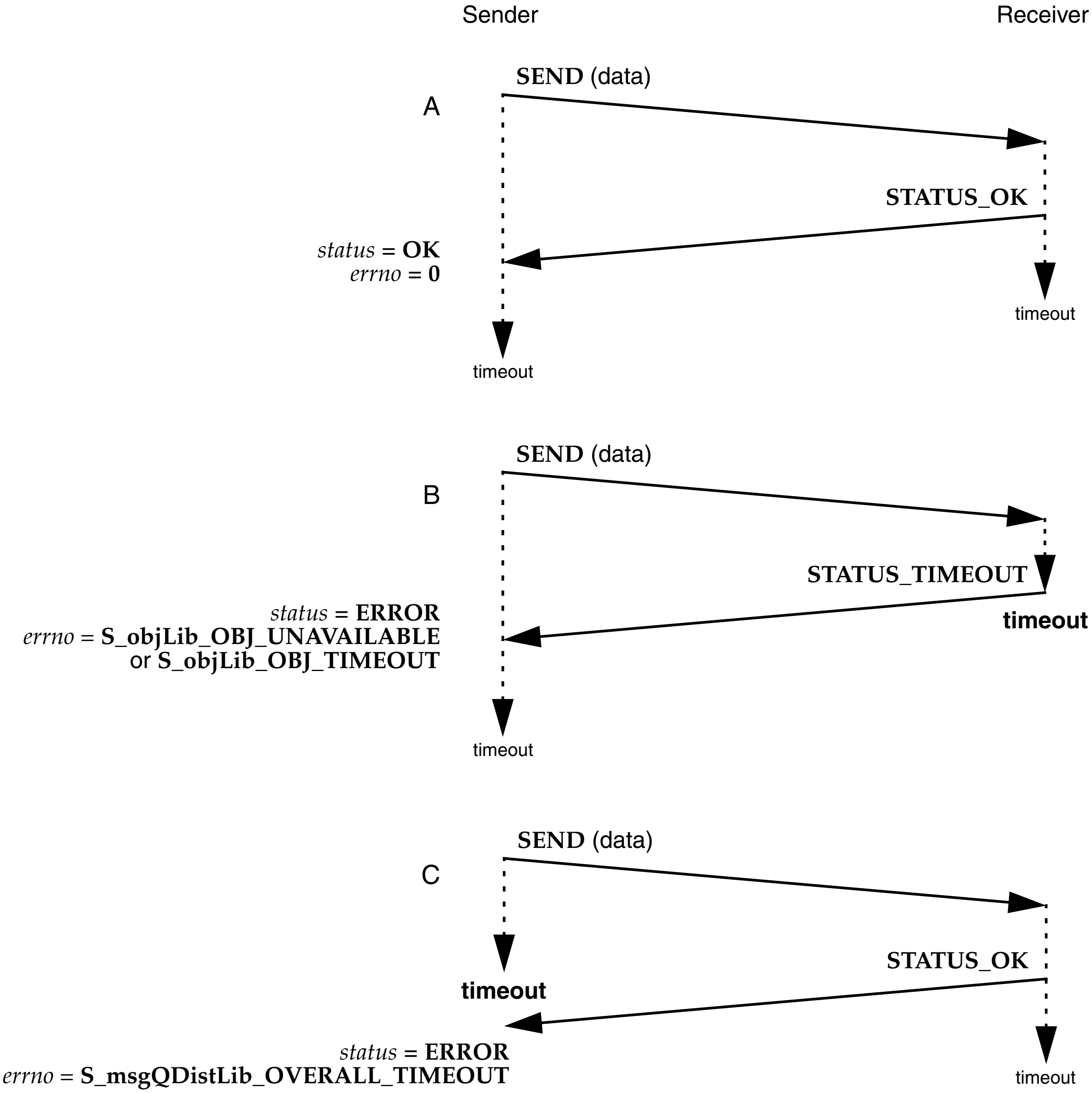
The timeout on the remote side is the *msgQTimeout* argument, the number of ticks to wait at the remote message queue. The local timeout is the *overallTimeout* argument, the number of ticks to wait overall, including the transmission time.

In Example A, no timeout occurs before the send action completes and the status is returned. Thus, the local node receives the OK and knows that the message was received. In B and C, one of the timeouts expires, the send routine returns ERROR, and the *errno* variable is set to indicate a timeout.

In B, the local node is aware of the timeout; however, in C, the local node times out before the status is received. In this case, the local node does not know whether or not the send completed. In Example C, the message has been added to the remote queue, even though the local operation failed, and the two nodes have different views of the state of the system. To avoid this problem, set *overallTimeout* to a value great enough that the status is always received.

Using **msgQSend( )** prevents a situation like Example C from occurring because **msgQSend( )** waits forever for a response from the remote side.

Figure 10-5 **Sending Scenarios**

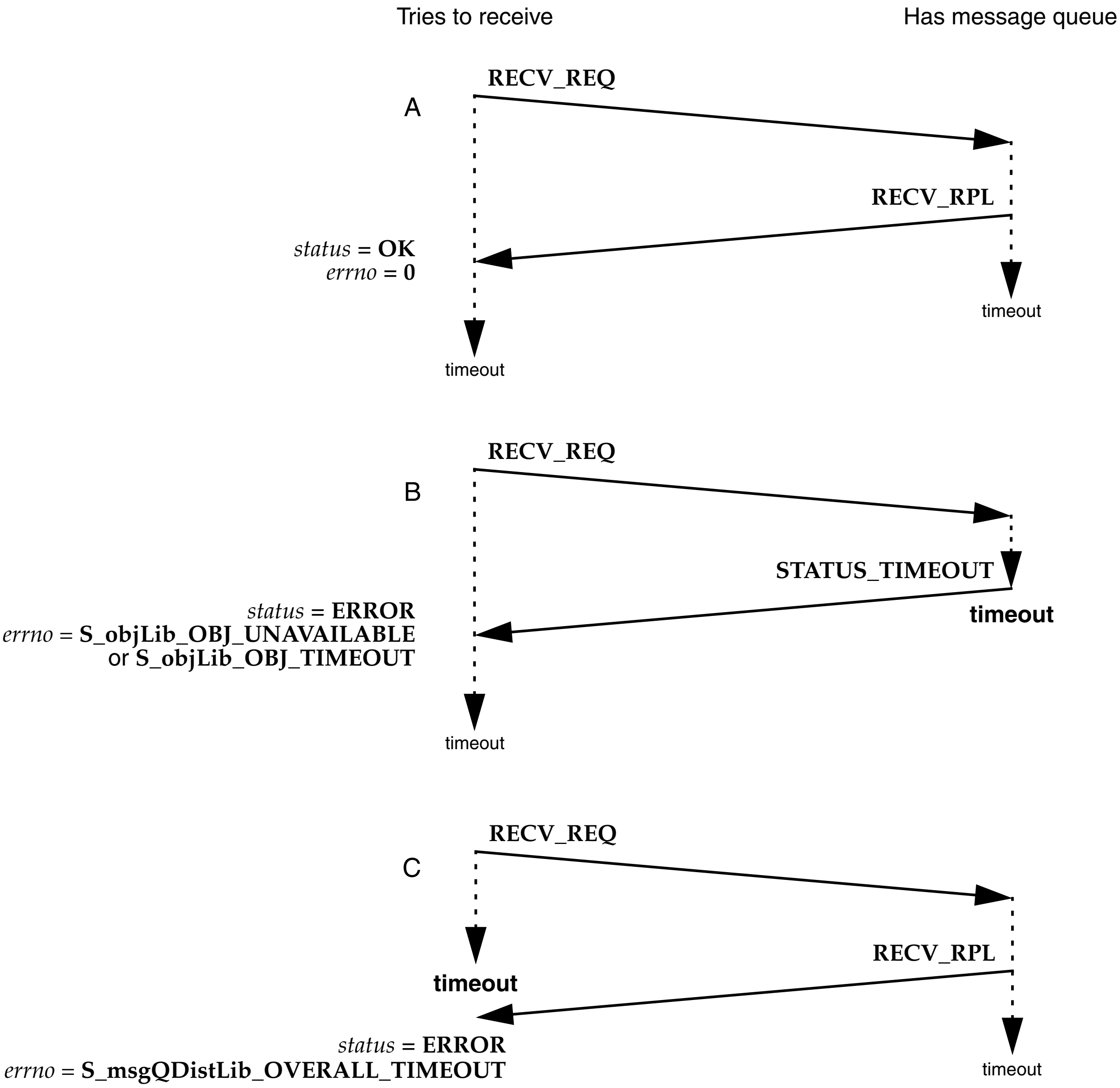


For limitations on sending messages to nodes that are unavailable, see *Detecting Absent Receiving Nodes*, p.418.

Receiving Limitations

As is the case for local sending of messages, local receiving occurs in the same manner for distributed message queues as for standard message queues.

Figure 10-6 **Receiving Scenarios**



As with the scenarios in *Sending Limitations*, p.410, different outcomes can result when receiving messages from remote message queues using the **msgQDistReceive( )** routine, depending on the value specified for the timeout arguments. Figure 10-6 presents three examples of messages being received by a node. There are two threads of execution: the local node waits for data to be received, and the remote node waits for data to arrive at the distributed message queue. Both threads are controlled with timeouts.

The timeout on the remote side is specified by the *msgQTimeout* argument, the number of ticks to wait at the remote message queue. The local timeout is specified by the *overallTimeout* argument, the number of ticks to wait overall, including the transmission time.

Example A illustrates a successful receive with no timeouts. A request to receive a message from a message queue is sent to the remote side and the result is returned before either thread experiences a timeout.

In B, the remote side experiences a timeout, but a status response is returned to the local node before the overall timeout expires. The receive routine returns an error and the *errno* variable is set to indicate a timeout. Both sides know that the receive failed and have the same view of the state of the remote message queue.

In C, the local node tries to receive a message from the remote node, but the *overallTimeout* expires before a response arrives. The local and remote sides end up with different views of the state of the remote message queue because, although the message was successfully removed from the queue on the remote side, the local side thinks the operation failed.

To avoid this problem, set the *overallTimeout* argument to a value great enough that the reply from the remote side is always received; or use **msgQReceive( )**, because it waits forever for a response from the remote side.

Displaying the Contents of Distributed Message Queues

To display the contents of a distributed message queue, use the standard message queue routine **msgQShow( )**.

The following example shows **msgQShow( )** output for a local distributed message queue:

```
[VxKernel]-> msgQShow 0xffe47f

Message Queue Id     : 0xffe47f
Global unique Id     : 0x930b267b:fe
Type                 : queue
Home Node            : 0x930b267b
Mapped to            : 0xea74d0
```

```
Message Queue Id    : 0xea74d0
Task Queueing       : FIFO
Message Byte Len    : 1024
Messages Max        : 100
Messages Queued     : 0
Receivers Blocked   : 0
Send Timeouts       : 0
Receive Timeouts    : 0
value = 0 = 0x0
```

The following example shows output for the same queue but from a different machine:

```
[VxKernel]-> msgQShow 0x3ff9b7

Message Queue Id    : 0x3ff9b7
Global unique Id    : 0x930b267b:fe
Type                : remote queue
Home Node           : 0x930b267b
value = 0 = 0x0
```

10.3.6 Working with Group Message Queues

VxFusion uses group message queues to support the multicasting of messages to a number of distributed message queues. A group message queue is a virtual message queue that takes any message sent to it and sends it to all member queues.

Table 10-9 lists the routines available to handle distributed group message queues.

Table 10-9 **Distributed Group Message Queue Routines**

Routines	Functionality
msgQDistGrpAdd( )	Adds a distributed message queue to a group.
msgQDistGrpDelete( )	Removes a message queue from a group.
msgQDistGrpShow( )	Displays information about a group message queue.

Group message queues are created and added to by the routine **msgQDistGrpAdd( )**. If a group message queue does not already exist when **msgQDistGrpAdd( )** is called, one is created, and the specified queue becomes the first member of the group. If a group message queue already exists, then the specified distributed message queue is simply added as a member. The **msgQDistGrpAdd( )** routine always returns the ID of the group message queue.

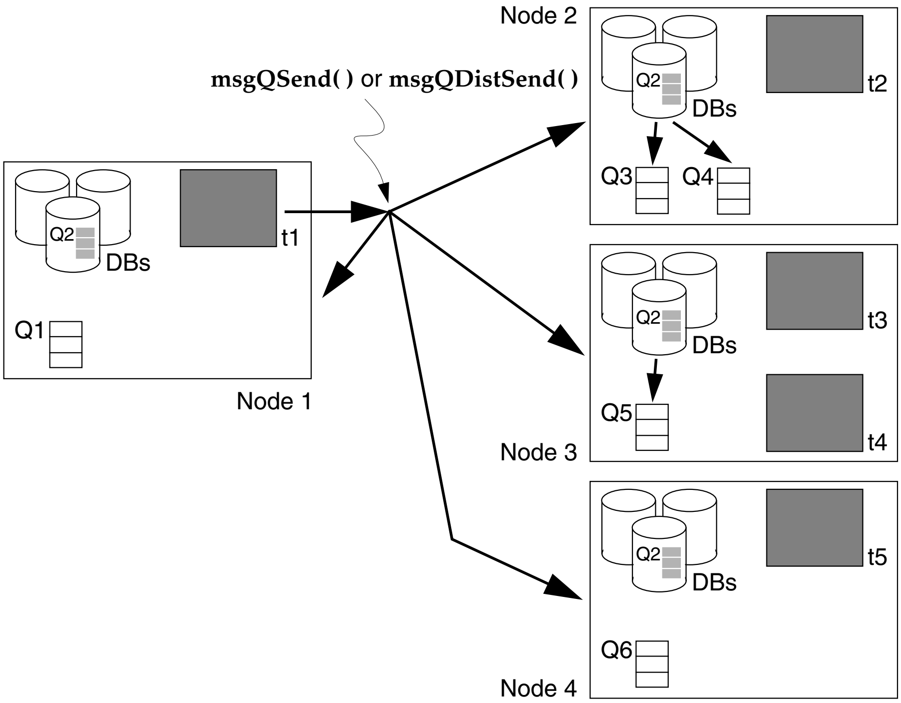
If you want a distributed message queue to belong to more than one group, you must call **msgQDistGrpAdd( )** to assert each additional membership.

Only sending to and displaying group message queues is supported. It is an error to try to receive from a group message queue or query it for the number of messages.

NOTE: Although there is a **msgQDistGrpDelete( )** routine, a distributed message queue cannot be deleted from a group message queue. The **msgQDistGrpDelete( )** command will always return ERROR.

Information about all the group message queues that have been created in the system and their locally added members is stored in the local copy of the distributed group database. The **msgQDistGrpShow( )** routine displays either all of the groups in the distributed group database along with their locally added members or a specific group and its locally added members. For more information on using **msgQDistGrpShow( )**, see *Displaying Information About Distributed Group Message Queues*, p.416.

Figure 10-7 **Group Message Queue**



Consider the example in Figure 10-7. The distributed message queues **Q1**, **Q3**, **Q4**, **Q5**, and **Q6** have all been previously created by calls to **msgQDistCreate( )**. They have been added to the distributed name database by calls to **distNameAdd( )**. The same tasks that created the message queues also added **Q3**, **Q4**, and **Q5** to the group message queue **Q2** by calling **msgQDistGrpAdd( )** for each new member. Message queues **Q1** and **Q6** have not been added to the **Q2** group. The first call to **msgQDistGrpAdd( )** creates **Q2** as an entry in each node's distributed group database. (In Figure 10-7, the three databases—distributed node, name, and group—are symbolized by three cylinders, the front one of which represents the group database and shows the **Q2** entry.)

NOTE: The distributed message queue members of **Q2** did not have to be added to the group by the task that created them. In fact, any task can add any distributed message queue to a group, as long as the ID for that queue is known locally or is available from the distributed name database.

In Figure 10-7, with group message queue **Q2** established, when task **t1** sends a message to the group **Q2**, the message is sent to all nodes in the system. Each node uses the distributed group database to identify group members and forwards the message to them. In this example, the message is sent to members **Q3**, **Q4**, and **Q5**, but not to non-members **Q1** and **Q6**.

For detailed information about distributed group message queues, see the related entries in the *VxWorks API Reference*.

Displaying Information About Distributed Group Message Queues

The **msgQDistGrpShow( )** routine displays either all of the groups in the group database along with their locally added members or a specific group and its locally added members.

The following output demonstrates the use of **msgQDistGrpShow( )** with no arguments:

```
[VxKernel]-> msgQDistGrpShow(0)

NAME OF GROUP         GROUP ID   STATE  MEMBER ID TYPE OF MEMBER
------------------- ---------- ------- ---------- --------------------------
grp1                  0x3ff9e3  global   0x3ff98b distributed msg queue
                                         0x3ff9fb distributed msg queue
grp2                  0x3ff933  global   0x3ff89b distributed msg queue
                                         0x3ff8db distributed msg queue
                                         0x3ff94b distributed msg queue
value = 0 = 0x0
```

The following call demonstrates the use of **msgQDistGrpShow( )** with the string "grp1" as the argument:

```
[VxKernel]-> msgQDistGrpShow("grp1")

NAME OF GROUP         GROUP ID   STATE  MEMBER ID TYPE OF MEMBER
------------------- ---------- ------- ---------- ---------------------------
grp1                  0x3ff9e3  global   0x3ff98b distributed msg queue
                                         0x3ff9fb distributed msg queue
value = 0 = 0x0
```

10.3.7 Working with Adapters

Adapters provide a uniform interface to VxFusion, regardless of the particular transport used. Table 10-10 presents the only API call related to adapters, **distIfShow( )**.

Table 10-10 **Adapter Routines**

Routine	Functionality
distIfShow( )	Displays information about the installed interface adapter.

For information on how to write your own VxFusion adapters, see *10.7 Designing Adapters*, p. 423. For detailed information about adapters, see the related entries in the *VxWorks API Reference*.

The following example demonstrates the use of **distIfShow( )**:

```
[VxKernel]-> distIfShow

Interface Name                 : "UDP adapter"
MTU                            : 1500
Network Header Size            : 14
SWP Buffer                     : 32
Maximum Number of Fragments    : 10
Maximum Length of Packet       : 14860
Broadcast Address              : 0x930b26ff
Telegrams received             : 23
Telegrams received for sending : 62
Incoming Telegrams discarded   : 0
Outgoing Telegrams discarded   : 0
```

To learn how to change the installed interface adapter or to modify its values, see *10.3.3 Configuring VxFusion*, p. 401.

10.4 System Limitations

Interrupt Service Routine Restrictions

Unlike standard message queues, distributed objects cannot be used at interrupt level. No routines that use distributed objects can be called from ISRs. An ISR is dedicated to handle time-critical processing associated with an external event; therefore, using distributed objects at interrupt time is not appropriate. On a multiprocessor system, run event-related time-critical processing on the CPU where the time-related interrupt occurred.

Detecting Absent Receiving Nodes

When a distributed message queue is created on one node, the other nodes are informed of its creation. However, if the node on which the message queue was created either crashes or is rebooted, there is no simple way for other nodes to detect the loss of the message queue. The entry in the name database is not modified, even if the system recreates the queue when it reboots. As a consequence, the other nodes might use an invalid message queue ID and pend on a receive notification that will never be sent. It is, therefore, the responsibility of the application on the receiving node to set timeouts when more data is expected.

10.5 Node Startup

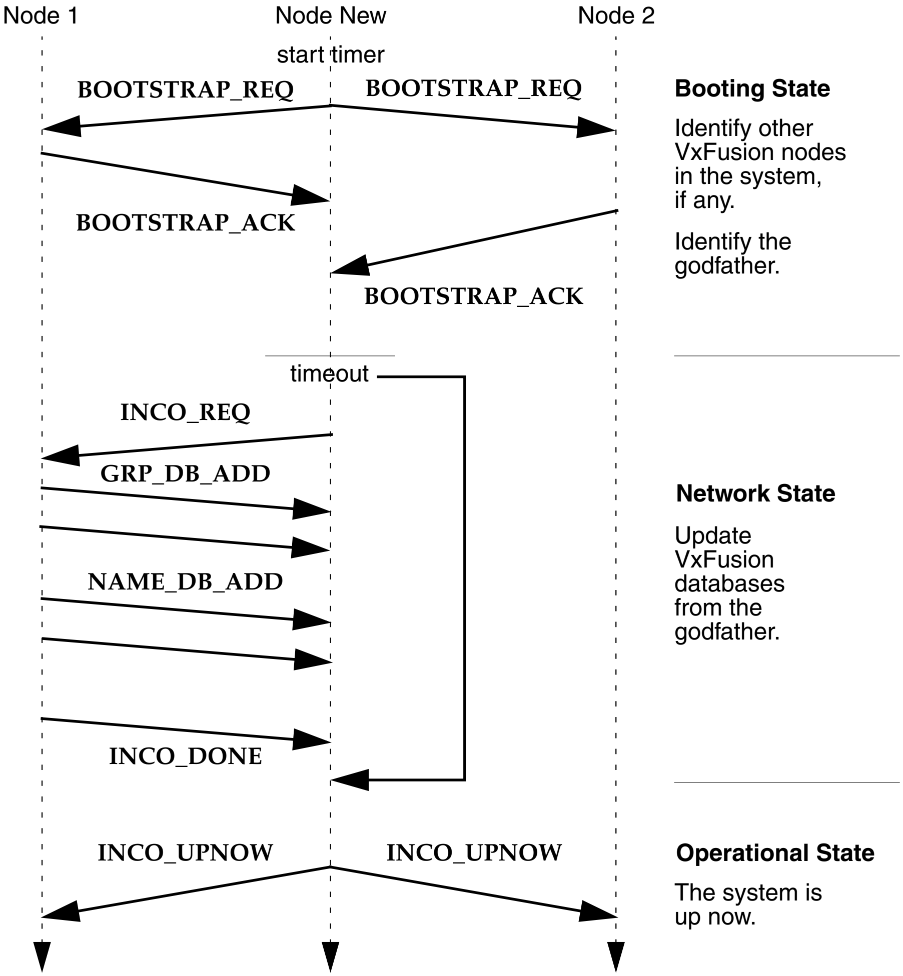
The VxFusion system is designed to support the addition of new nodes at run-time, as well as to survive the failure of one or more nodes. The ability to add new nodes at run-time is made possible by the node startup (incorporation) process. The ability to survive the failure of one or more nodes is made possible by replicated databases that are populated during the node startup process. This section discusses node startup in detail.

Table 10-11 **Node Startup States**

State	Activities
Booting	Identify other nodes in the system and determine the “godfather.”
Network	Update databases from the “godfather.”
Operational	Notify the other nodes that the node is up and running.

Table 10-11 lists its three states, each of which is described in detail in this section. Figure 10-8 illustrates the node startup process. (To simplify Figure 10-8, it does not show the sending of acknowledgments.)

Figure 10-8 **Starting Up the System**



Booting State

When VxFusion is first initialized on a node, it broadcasts a bootstrap request message (**BOOTSTRAP_REQ**), which is used to locate other active VxFusion nodes on the network. Nodes that receive the bootstrap request message respond with a bootstrap acknowledgment message (**BOOTSTRAP_ACK**).

The node that sends the first bootstrap acknowledgment response received by another node becomes the *godfather* for the local node. In Figure 10-8, Node 1 is the godfather because its bootstrap acknowledgment is received first. The purpose of the godfather is to help the local node update its databases. If no response is received within a specified period of time, the node assumes it is the first node to come up on the network.

As soon as a godfather is located or as soon as the assumption is made that a node is first in the network, the node shifts from the *booting state* to the *network state*.

Network State

Once a godfather is located, the local node asks the godfather to update its databases by sending an incorporation request message (**INCO_REQ**). The godfather updates the local node's name and group databases. These updates are indicated by the **GRP_DB_ADD** and **NAME_DB_ADD** arrows in Figure 10-8. The godfather tells the receiving node that it is finished updating the databases by sending an incorporation done message (**INCO_DONE**).

Once the database updates have completed, the node moves into the *operational state*. If there is no godfather, the node moves directly from the booting state to the operational state.

Operational State

When a node moves into the operational state, VxFusion is fully initialized and running on it. The node broadcasts the “up now” incorporation message (**INCO_UPNOW**) to tell the other nodes in the system that it is now active.

10.6 Telegrams and Messages

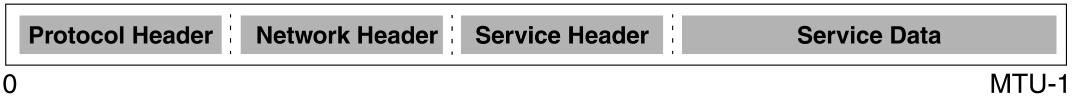
10.6.1 Telegrams Versus Messages

Because VxFusion is sending data over a network, the total byte size of a single message may be too great to transmit as a single unit. It may have to be broken into smaller segments. These VxFusion message segments are called *telegrams*. Telegrams are the largest *packets* that can be sent between nodes.

A telegram is an atomic transmission on the transport of a certain number of bytes. The MTU size of the transport, which is provided to VxFusion by the adapter at initialization time, defines the size of the telegram.

When you send a message, the system segments it into one or more telegrams. Figure 10-9 shows a telegram and its parts:

Figure 10-9 **A Telegram**



Protocol Header

The transport defines the protocol header, which it builds from values provided by the adapter. The contents of this header vary from protocol to protocol, but may include fields such as source address, destination address, and priority, if the transport supports priority.

Network Header

The adapter defines and builds the network header. See *10.7.1 Designing the Network Header*, p.424 for a detailed description of the network header and its fields.

Service Header

VxFusion defines and builds the service header. The service header is a small header that identifies the internal service sending the message and the message type.

Service Data

The service data is the data being sent. When sending a message to a remote message queue or a message queue group, this data can be either the entire message or a fragment of the message to be sent. It is a fragment of the message if the message size exceeds the space allocated in the telegram for service data.

Because large numbers of telegrams can be necessary when working with transports having small MTU sizes, VxFusion does not acknowledge individual telegrams. Instead, only whole messages are acknowledged. In the event of a transmission error or if a telegram is lost, the whole message must be re-transmitted.

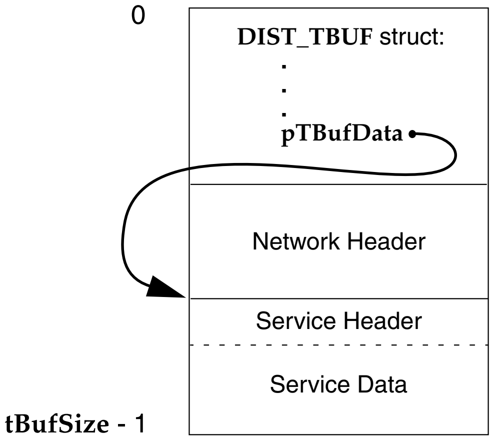
10.6.2 Telegram Buffers

When VxFusion needs to send a message to a remote node, it fragments the message into telegram buffers allocated from a pool of pre-allocated telegram buffers. Figure 10-10 shows a telegram buffer and its parts.

VxFusion sends one telegram buffer at a time to the adapter, which constructs a corresponding telegram from the network header, service header, and service data in the buffer. Finally, the adapter sends out the telegram.

On the receiving node, when the adapter receives a telegram, it is responsible for reconstructing the telegram buffer from the telegram. After reconstructing the telegram buffer, the adapter sends it to VxFusion.

Figure 10-10 **A Telegram Buffer**



The **DIST_TBUF** structure member, **pTBufData**, contains the address of the start of the service header. To access the network header, subtract the size of the network header from this address.

In order to reconstruct a telegram buffer at the remote node, some telegram buffer fields must be copied into the network header by the adapter before sending the telegram. For a list of the fields that must be copied into the network header, see *10.7.1 Designing the Network Header*, p.424.

10.7 Designing Adapters

This section describes how to write adapters for the VxFusion component.

An adapter is a software mechanism that facilitates communication between VxFusion nodes. An adapter sits between VxFusion and the single communications transport, as shown in Figure 10-1. A transport can be a high-level communications protocol, such as UDP, or a low-level communications or bus driver, such as an Ethernet driver. When a message is sent to a remote node, VxFusion passes a telegram buffer to the adapter, and the adapter sends data to the remote node by way of the supported transport. Similarly, when a message is received from a remote node, an adapter reconstructs a telegram buffer from the incoming data and sends the buffer to VxFusion.

An adapter must provide the following:

- a network header to transmit telegram buffer fields
- an initialization function
- a startup routine
- a send function
- an input function
- an I/O control function

Each of these items is described in more detail in this section.

NOTE: In this release, VxFusion supports only one adapter and, thus, only one transport.

10.7.1 Designing the Network Header

The telegram buffer itself is not passed to the remote node; it goes from the local node only as far as the adapter. To move data from the adapter to a remote node, VxFusion uses the adapter's network header, a structure that stores data from certain telegram buffer fields, data that is required to reconstruct the telegram buffer at the remote node. It is the network header and the data pointed to by the telegram buffer that pass from the adapter to the remote node.

Performance and message size are directly related to the size of the network header. Because you design the network header, you can influence performance by specifying the portion of the telegram dedicated to the network header. You trade off throughput for message size.

For example, for a transport with a small transmission unit size, it may be desirable to use a small network header to maximize throughput. However, reducing the number of bits for a field, like the fragment sequence number, also reduces the size of the message that can be sent. On the other hand, for transports having large transmission unit sizes, the network header constitutes a smaller percentage of the overall telegram; the adapter can use a larger fragment sequence number in the header without affecting throughput performance significantly.

The telegram buffer structure, **DIST_TBUF**, is as follows:

```
typedef struct  /* DIST_TBUF */
    {
    DIST_TBUF_GEN          tBufGen;    /* TBufGen struct */
    void                   *pTBufData; /* pointer to the data */

/* Fields required to construct the telegram buffer on the remote side */
    uint16_t               tBufId;     /* ID of the packet */
    uint16_t               tBufAck;    /* ID of packet last received and */
                                       /* ackowledged without error */
    uint16_t               tBufSeq;    /* sequence number of the fragment */
    uint16_t               tBufNBytes; /* number of non-network header data
                                       /* bytes */
    uint16_t               tBufType;   /* type of telegram */
    uint16_t               tBufFlags;  /* telegrams flags */
    } DIST_TBUF;
```

You must create fields in the adapter-specific network header that correspond to the following **DIST_TBUF** fields. All but the first two fields, *tBufGen* and *pTBufData*, are required to reconstruct the telegram buffer on the remote side:

You may need to create and add fields to the network header depending on the transport. For example, the message priority is an argument to both the **distIfXxxSend( )** and **distNetInput( )** routines, but not all transports support priority. If the transport supports priority, the priority is transmitted in the protocol header and is available at the remote side. If the transport does not support priority and you want it preserved, then you should add a field to the network header to transmit the priority value to the remote node.

10.7.2 Writing an Initialization Routine

An adapter is initialized by a series of automatic calls to initialization routines. You write some of the code that accomplishes this.

The **distInit( )** routine, which initializes the whole of VxFusion, is called automatically when a target is booted with a VxWorks image that has VxFusion installed. The prototype for **distInit( )** follows:

```
STATUS distInit
    (
    DIST_NODE_ID   myNodeId,       /* node ID of this node */
    FUNCPTR        ifInitRtn,      /* interface init routine */
    void           *pIfInitConf,   /* ptr to interface configuration */
    int            maxTBufsLog2,   /* max number of telegram buffers */
    int            maxNodesLog2,   /* max number of nodes in node db */
    int            maxQueuesLog2,  /* max number of queues on this node */
    int            maxGroupsLog2,  /* max number of groups in db */
    int            maxNamesLog2,   /* max bindings in name db */
    int            waitNTicks      /* wait in ticks when bootstrapping */
    )
    {
    }
```

The argument *ifInitRtn* specifies the adapter initialization routine, **distIf***Xxx***Init( )**, where *Xxx* is the name of the adapter specified by you.

NOTE: Never call **distInit( )** or **distIf***Xxx***Init( )** directly.

You can base your version of **distIf***Xxx***Init( )** on the following prototypical code:

```
STATUS distIfXxxInit
    (
    void     *pConf,     /* ptr to configuration data, if any */
    FUNCPTR  *pStartup   /* ptr to startup routine */
    );
```

distIf*Xxx***Init( )** should be invoked with the following parameters:

pConf
a pointer to interface configuration data pointed to by the *pIfInitConf* argument of **distInit( )**

pStartup
a pointer to a startup routine that is set up after the adapter initialization routine returns

The adapter initialization routine should perform the following operations:

- set the startup routine pointer to point to the adapter startup routine
- set the fields of the **DIST_IF** structure

The **DIST_IF** structure is one of the mechanisms that provides VxFusion with its transport independence: the structure maintains its form regardless of the adapter being used. The **DIST_IF** structure is composed of configurable fields that identify the adapter to be used and operating conditions for sending messages. For information about using **DIST_IF**, see *Using the DIST_IF Structure*, p.426.

Although you cannot call **distInit( )** and **distIf***Xxx***Init( )** directly, you can modify the VxFusion startup code in **usrVxFusion.c** to change the VxFusion initialization process.

If you need to specify additional information to initialize the adapter, you can modify the *pIfInitConf* argument of **distInit( )** to provide that information. The *pIfInitConf* argument is passed as the *pConf* argument of **distIf***Xxx***Init( )**. To preserve information pointed to by *pConf*, you should copy its values into a more permanent structure within the adapter. If the adapter needs no additional configuration information, then *pConf* should be ignored.

The adapter initialization routine should return OK if initialization is successful, or ERROR if it fails.

Using the DIST_IF Structure

Use the **DIST_IF** structure to pass information about the adapter to VxFusion so VxFusion can fragment messages into telegrams of appropriate size to be sent out over the transport.

The **DIST_IF** structure has the following declaration:

```
typedef struct   /* DIST_IF */
    {
    char         *distIfName;         /* name of the interface */
    int          distIfMTU;           /* MTU size of interface's transport */
    int          distIfHdrSize;       /* network header size */
    DIST_NODE_ID distIfBroadcastAddr; /* broadcast addr for the interface */
    short        distIfRngBufSz;      /* # bufs in sliding window protocol */
    short        distIfMaxFrags;      /* max frags msg can be broken into */
    int      (*distIfIoctl) (int fnc, ...); /* adapter IOCTL function */
    STATUS   (*distIfSend) (DIST_NODE_ID destId, DIST_TBUF *pTBuf, int prio);
                                           /* send function of the adapter */
    } DIST_IF;
```

Fields for **DIST_IF** are defined as follows:

distIfName
This field is the name of the interface or adapter. It is displayed when **distIfShow( )** is called.

distIfMTU
This field specifies the MTU size of the transport.

distIfHdrSize
This field specifies the size of the adapter-specific network header. Some adapters may use smaller headers, if their MTU size is small, to maximize the amount of data per telegram. Adapters for transports having a large MTU size may use a larger header to allow larger messages.

distIfBroadcastAddr
This field specifies the broadcast address for the transport. If the transport does not support the broadcast operation, then a dummy broadcast address must be provided and the transport must simulate the broadcast operation whenever it receives a message destined for the dummy broadcast address. One way of simulating the broadcast operation is to send the message to all other nodes in the system using point-to-point addressing.

distIfRngBufSz
This field specifies the number of elements to use in the sliding window protocol. A larger number means the greater the number of telegrams that can be held in the ring buffer for messages awaiting acknowledgment.

distIfMaxFrags
This field specifies the maximum number of fragments into which a message can be broken. The maximum size of a message that can be sent is:

maximum size = (number of fragments) x (MTU size - network header size - service header size)

The size of the service header depends on the type of VxFusion message being sent, for example, **BOOTSTRAP_REQ** or **BOOTSTRAP_ACK**.

distIfIoctl
This is the **ioctl** routine for the adapter interface.

distIfSend
This is the send routine for the adapter interface.

10.7.3 Writing a Startup Routine

The adapter startup routine should be returned to **distInit( )** by the adapter initialization routine. You can use the following prototype to write a startup routine for adapter interface *Xxx*:

```
STATUS distXxxStart
    (
    void *   pConf    /* ptr to configuration data */
    );
```

The startup routine is invoked by **distInit( )** after the network layer of VxFusion is initialized. The startup routine should spawn the input task, as well as any initialization or startup that must be done to enable the transmission and reception of telegrams over the desired transport. For example, at this time, a UDP adapter should create the socket that it uses for communications.

The startup routine should return OK if the operation is successful, or ERROR if it fails.

10.7.4 Writing a Send Routine

You can write a send routine, **disIf***Xxx***Send( )**, for adapter interface *Xxx* based on the following prototypical code:

```
STATUS distIfXxxSend
    (
    DIST_NODE_ID   nodeIdDest,  /* destination node */
    DIST_TBUF      pTBuf,       /* TBUF to send */
    int            priority     /* packet priority */
    );
```

Arguments for **disIf***Xxx***Send( )** should be defined as follows:

nodeIdDest
    This parameter is the unique identifier of the destination node or the broadcast node.

pTBuf
    This parameter is the buffer that gets sent out.

priority
    This parameter states the priority of the message. It may or not be supported by the transport.

The purpose of the send routine is to take a telegram buffer passed from VxFusion and send the associated telegram out over the transport.

The send routine should perform the following tasks:

- Increment the statistics (**distStat.ifOutReceived++**).
- Locate and fill in the pre-allocated network header with values from the telegram buffer that is passed in as an argument. The network header fields should be filled in using network byte order, so they can be correctly decoded on the remote side, even if the remote node uses a different byte order. (For information about which telegram buffer fields must be copied, see *10.7.1 Designing the Network Header*, p. 424.)
- Fill in any additional network header fields (such as *priority*) that may need to be filled in.
- Send the telegram.

The send routine should return OK if the operation is successful, or ERROR if it fails.

10.7.5 Writing an Input Routine

You can write an input routine, **disIf***Xxx***InputTask( )**, for adapter interface *Xxx* based on the following prototypical code:

```
void distIfXxxInputTask( );
```

The purpose of the input routine is to read a telegram and send it to VxFusion by calling **distNetInput( )**. For more information, see the entry for **distNetInput( )** in the *VxWorks API Reference*.

The input routine should listen or wait for an incoming telegram from the transport. Upon receipt of the telegram, the statistics field **distStat.ifInReceived** should be incremented. Then, the telegram should be tested to make sure that it is longer than the network header. If not, the telegram is too small and should be ignored. The **distStat.ifInLength** and **distStat.ifInDiscarded** fields should also be incremented, in this case.

If your transport does not discard broadcast packets sent from itself, use the input routine to filter out transport-originated broadcast packets.

After the input routine has discarded any duplicate or faulty telegram that originates from the transport, the incoming telegram is assumed to be correct (although there is one more check made later). A telegram buffer is allocated and the contents of the telegram are copied into it. The non-data fields of the telegram buffer that were not transmitted are as follows:

- *tBufId*
- *tBufAck*
- *tBufSeq*
- *tBufNBytes*
- *tBufType*
- *tBufFlags*

These fields are reconstructed using the network header. During the reconstruction, these fields should be converted back to host order from network order.

After the telegram buffer is reconstructed and the number of bytes expected in the non-header portion of the telegram are known (from the *tBufNBytes* field), telegram length is compared to *tBufNBytes* plus the size of the network header. If the lengths do not match, the telegram should be discarded and the statistics **distStat.ifInLength** and **distStat.ifInDiscarded** incremented.

If the lengths match, the telegram should be forwarded upward by calling **distNetInput( )**.

10.7.6 Writing an I/O Control Routine

You can write an I/O control routine, **distIf***Xxx***Ioctl( )**, for adapter interface *Xxx* based on the following prototypical code:

```
int distIfXxxIoctl(int func, …);
```

You determine the control functions that must be supported by the adapter.

This routine should return the return value of the I/O control function being performed if successful, or ERROR if the operation fails. If no control functions are provided, the **distIf***Xxx***Ioctl( )** routine should return ERROR.

11
Shared-Memory Objects
Optional Component VxMP

11.1 Introduction

VxMP is an optional VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running on separate CPUs. For information on how to install VxMP, see *Tornado Getting Started*.

Shared-memory objects are a class of system objects that can be accessed by tasks running on different processors. They are called *shared-memory* objects because the object's data structures must reside in memory accessible by all processors. Shared-memory objects are an extension of local VxWorks objects. *Local objects* are only available to tasks on a single processor. VxMP supplies three kinds of shared-memory objects:

- shared semaphores (binary and counting)
- shared message queues
- shared-memory partitions (system- and user-created partitions)

Shared-memory objects provide the following advantages:

- A transparent interface that allows shared-memory objects to be manipulated with the same routines that are used for manipulating local objects.
- High-speed inter-processor communication—no unnecessary packet passing is required.
- The shared memory can reside either in dual-ported RAM or on a separate memory board.

The components of VxMP consist of the following: a name database (**smNameLib**), shared semaphores (**semSmLib**), shared message queues (**msgQSmLib**), and a shared-memory allocator (**smMemLib**).

This chapter presents a detailed description of each shared-memory object and internal considerations. It then describes configuration and troubleshooting.

11.2 Using Shared-Memory Objects

VxMP provides a transparent interface that makes it easy to execute code using shared-memory objects on both a multiprocessor system and a single-processor system. After an object is created, tasks can operate on shared objects with the same routines used to operate on their corresponding local objects. For example, shared semaphores, shared message queues, and shared-memory partitions have the same syntax and interface as their local counterparts. Routines such as **semGive( )**, **semTake( )**, **msgQSend( )**, **msgQReceive( )**, **memPartAlloc( )**, and **memPartFree( )** operate on both local and shared objects. Only the create routines are different. This allows an application to run in either a single-processor or a multiprocessor environment with only minor changes to system configuration, initialization, and object creation.

All shared-memory objects can be used on a single-processor system. This is useful for testing an application before porting it to a multiprocessor configuration. However, for objects that are used only locally, local objects always provide the best performance.

After the shared-memory facilities are initialized (see *11.4 Configuration*, p. 454 for initialization differences), all processors are treated alike. Tasks on any CPU can create and use shared-memory objects. No processor has priority over another from a shared-memory object's point of view.[1]

Systems making use of shared memory can include a combination of supported architectures. This enables applications to take advantage of different processor types and still have them communicate. However, on systems where the processors have different byte ordering, you must call the macros **ntohl** and **htonl** to byte-swap the application's shared data (see *VxWorks Network Programmer's Guide: TCP/IP Under VxWorks*).

[1] Do not confuse this type of priority with the CPU priorities associated with VMEbus access.

When an object is created, an *object ID* is returned to identify it. For tasks on different CPUs to access shared-memory objects, they must be able to obtain this ID. An object's ID is the same regardless of the CPU. This allows IDs to be passed using shared message queues, data structures in shared memory, or the name database.

Throughout the remainder of this chapter, system objects under discussion refer to shared objects unless otherwise indicated.

11.2.1 Name Database

The *name database* allows the association of any value to any name, such as a shared-memory object's ID with a unique name. It can communicate or *advertise* a shared-memory block's address and object type. The name database provides name-to-value and value-to-name translation, allowing objects in the database to be accessed either by name or by value. While other methods exist for advertising an object's ID, the name database is a convenient method for doing this.

Typically the task that creates an object also advertises the object's ID by means of the name database. By adding the new object to the database, the task associates the object's ID with a name. Tasks on other processors can look up the name in the database to get the object's ID. After the task has the ID, it can use it to access the object. For example, task **t1** on CPU 1 creates an object. The object ID is returned by the creation routine and entered in the name database with the name **myObj**. For task **t2** on CPU 0 to operate on this object, it first finds the ID by looking up the name **myObj** in the name database.

Table 11-1 **Name Service Routines**

Routine	Functionality
smNameAdd( )	Adds a name to the name database.
smNameRemove( )	Removes a name from the name database.
smNameFind( )	Finds a shared symbol by name.
smNameFindByValue( )	Finds a shared symbol by value.
smNameShow( )	Displays the name database to the standard output device.*

* Automatically included if **INCLUDE_SM_OBJ** is selected.

This same technique can be used to advertise a shared-memory address. For example, task **t1** on CPU 0 allocates a chunk of memory and adds the address to the database with the name **mySharedMem**. Task **t2** on CPU 1 can find the address of this shared memory by looking up the address in the name database using **mySharedMem**.

Tasks on different processors can use an agreed-upon name to get a newly created object's value. See Table 11-1 for a list of name service routines. Note that retrieving an ID from the name database need occur only one time for each task, and usually occurs during application initialization.

The name database service routines automatically convert to or from network-byte order; do not call **htonl( )** or **ntohl( )** explicitly for values from the name database.

The object types listed in Table 11-2 are defined in **smNameLib.h**.

Table 11-2 **Shared-Memory Object Types**

Constant	Hex Value
T_SM_SEM_B	0
T_SM_SEM_C	1
T_SM_MSG_Q	2
T_SM_PART_ID	3
T_SM_BLOCK	4

The following example shows the name database as displayed by **smNameShow( )**, which is automatically included if **INCLUDE_SM_OBJ** is selected for inclusion in the project facility VxWorks view. The parameter to **smNameShow( )** specifies the level of information displayed; in this case, 1 indicates that all information is shown. For additional information on **smNameShow( )**, see its reference entry.

```
-> smNameShow 1
value = 0 = 0x0
```

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

```
Name in Database Max : 100 Current : 5 Free : 95
Name                   Value           Type
----------------- ------------- -------------
myMemory             0x3835a0       SM_BLOCK
myMemPart            0x3659f9       SM_PART_ID
```

```
myBuff              0x383564       SM_BLOCK
mySmSemaphore       0x36431d       SM_SEM_B
myMsgQ              0x365899       SM_MSG_Q
```

11.2.2 Shared Semaphores

Like local semaphores, *shared semaphores* provide synchronization by means of atomic updates of semaphore state information. See *2. Basic OS* in this manual and the reference entry for **semLib** for a complete discussion of semaphores. Shared semaphores can be given and taken by tasks executing on any CPU with access to the shared memory. They can be used for either synchronization of tasks running on different CPUs or mutual exclusion for shared resources.

To use a shared semaphore, a task creates the semaphore and advertises its ID. This can be done by adding it to the name database. A task on any CPU in the system can use the semaphore by first getting the semaphore ID (for example, from the name database). When it has the ID, it can then take or give the semaphore.

In the case of employing shared semaphores for mutual exclusion, typically there is a system resource that is shared between tasks on different CPUs and the semaphore is used to prevent concurrent access. Any time a task requires exclusive access to the resource, it takes the semaphore. When the task is finished with the resource, it gives the semaphore.

For example, there are two tasks, **t1** on CPU 0 and **t2** on CPU 1. Task **t1** creates the semaphore and advertises the semaphore's ID by adding it to the database and assigning the name **myMutexSem**. Task **t2** looks up the name **myMutexSem** in the database to get the semaphore's ID. Whenever a task wants to access the resource, it first takes the semaphore by using the semaphore ID. When a task is done using the resource, it gives the semaphore.

In the case of employing shared semaphores for synchronization, assume a task on one CPU must notify a task on another CPU that some event has occurred. The task being synchronized pends on the semaphore waiting for the event to occur. When the event occurs, the task doing the synchronizing gives the semaphore.

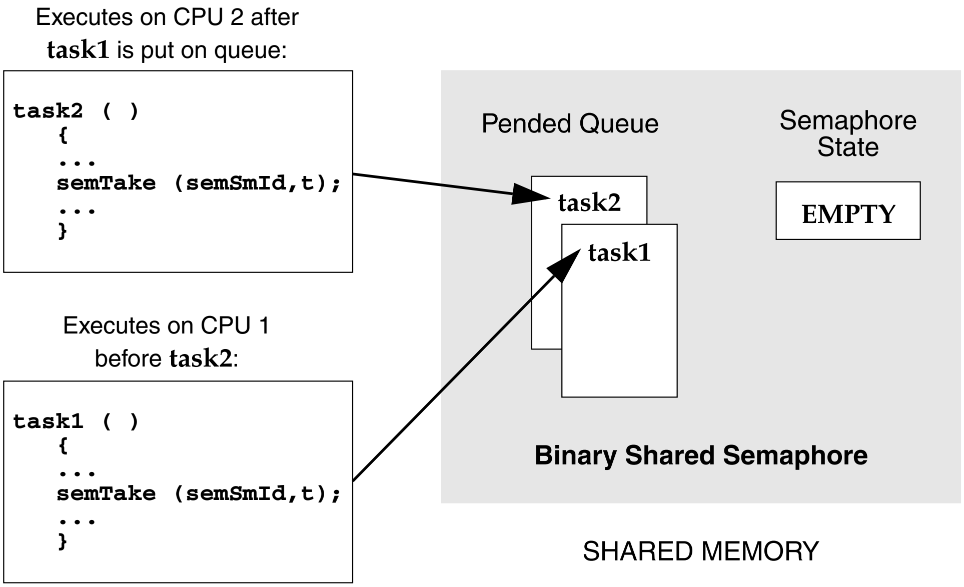
For example, there are two tasks, **t1** on CPU 0 and **t2** on CPU 1. Both **t1** and **t2** are monitoring robotic arms. The robotic arm that is controlled by **t1** is passing a physical object to the robotic arm controlled by **t2**. Task **t2** moves the arm into position but must then wait until **t1** indicates that it is ready for **t2** to take the object. Task **t1** creates the shared semaphore and advertises the semaphore's ID by adding it to the database and assigning the name **objReadySem**. Task **t2** looks up the name **objReadySem** in the database to get the semaphore's ID. It then takes the semaphore by using the semaphore ID. If the semaphore is unavailable, **t2** pends, waiting for **t1** to indicate that the object is ready for **t2**. When **t1** is ready to transfer control of the object to **t2**, it gives the semaphore, readying **t2** on CPU1.

Table 11-3 **Shared Semaphore Create Routines**

Create Routine	Description
semBSmCreate( )	Creates a shared binary semaphore.
semCSmCreate( )	Creates a shared counting semaphore.

There are two types of shared semaphores, binary and counting. Shared semaphores have their own create routines and return a **SEM_ID**. Table 11-3 lists the create routines. All other semaphore routines, except **semDelete( )**, operate transparently on the created shared semaphore.

Figure 11-1 **Shared Semaphore Queues**



```
task2 ( )
   {
   ...
   semTake (semSmId,t);
   ...
   }
```

```
task1 ( )
   {
   ...
   semTake (semSmId,t);
   ...
   }
```

The use of shared semaphores and local semaphores differs in several ways:

- The shared semaphore queuing order specified when the semaphore is created must be FIFO. Figure 11-1 shows two tasks executing on different CPUs, both trying to take the same semaphore. Task 1 executes first, and is put at the front of the queue because the semaphore is unavailable (empty). Task 2 (executing on a different CPU) tries to take the semaphore after task 1's attempt and is put on the queue behind task 1.

- Shared semaphores *cannot* be given from interrupt level.
- Shared semaphores cannot be deleted. Attempts to delete a shared semaphore return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Use **semInfo( )** to get the shared task control block of tasks pended on a shared semaphore. Use **semShow( )**, if **INCLUDE_SEM_SHOW** is included in the project facility VxWorks view, to display the status of the shared semaphore and a list of pended tasks. The following example displays detailed information on the shared semaphore **mySmSemaphoreId** as indicated by the second argument (0 = summary, 1 = details):

```
-> semShow mySmSemaphoreId, 1
value = 0 = 0x0
```

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

```
Semaphore Id     : 0x36431d
Semaphore Type   : SHARED BINARY
Task Queuing     : FIFO
Pended Tasks     : 2
State            : EMPTY
TID            CPU Number      Shared TCB
------------- ------------- --------------
0xd0618            1           0x364204
0x3be924           0           0x36421c
```

Example 11-1 **Shared Semaphores**

The following code example depicts two tasks executing on different CPUs and using shared semaphores. The routine **semTask1( )** creates the shared semaphore, initializing the state to full. It adds the semaphore to the name database (to enable the task on the other CPU to access it), takes the semaphore, does some processing, and gives the semaphore. The routine **semTask2( )** gets the semaphore ID from the database, takes the semaphore, does some processing, and gives the semaphore.

```
/* semExample.h - shared semaphore example header file */

#define SEM_NAME "mySmSemaphore"

/* semTask1.c - shared semaphore example */

/* This code is executed by a task on CPU #1 */
#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
```

```
#include "smNameLib.h"
#include "stdio.h"
#include "taskLib.h"
#include "semExample.h"

/*
 * semTask1 - shared semaphore user
 */

STATUS semTask1 (void)
    {
    SEM_ID semSmId;

    /* create shared semaphore */

    if ((semSmId = semBSmCreate (SEM_Q_FIFO, SEM_FULL)) == NULL)
        return (ERROR);

    /* add object to name database */

    if (smNameAdd (SEM_NAME, semSmId, T_SM_SEM_B) == ERROR)
        return (ERROR);

    /* grab shared semaphore and hold it for awhile */

    semTake (semSmId, WAIT_FOREVER);

    /* normally do something useful */

    printf ("Task1 has the shared semaphore\n");
    taskDelay (sysClkRateGet () * 5);
    printf ("Task1 is releasing the shared semaphore\n");

    /* release shared semaphore */

    semGive (semSmId);

    return (OK);
    }

/* semTask2.c - shared semaphore example */

/* This code is executed by a task on CPU #2. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"

#include "smNameLib.h"
#include "stdio.h"
#include "semExample.h"

/*
 * semTask2 - shared semaphore user
 */
```

```
STATUS semTask2 (void)
    {
    SEM_ID semSmId;
    int    objType;

    /* find object in name database */

    if (smNameFind (SEM_NAME, (void **) &semSmId, &objType, WAIT_FOREVER)
        == ERROR)
        return (ERROR);

    /* take the shared semaphore */

    printf ("semTask2 is now going to take the shared semaphore\n");
    semTake (semSmId, WAIT_FOREVER);

    /* normally do something useful */

    printf ("Task2 got the shared semaphore!!\n");

    /* release shared semaphore */

    semGive (semSmId);

    printf ("Task2 has released the shared semaphore\n");

    return (OK);
    }
```

11.2.3 Shared Message Queues

Shared message queues are FIFO queues used by tasks to send and receive variable-length messages on any of the CPUs that have access to the shared memory. They can be used either to synchronize tasks or to exchange data between tasks running on different CPUs. See *2. Basic OS* in this manual and the reference entry for **msgQLib** for a complete discussion of message queues.

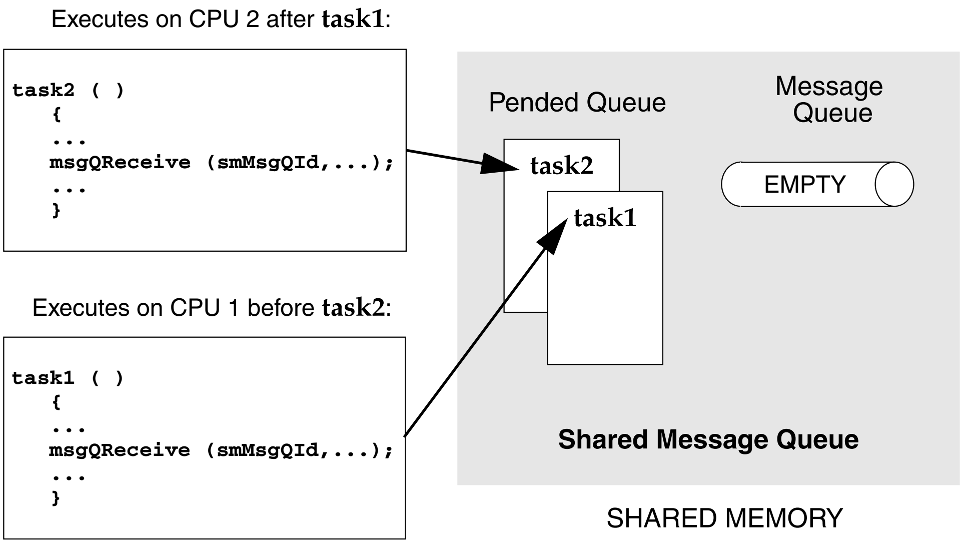
To use a shared message queue, a task creates the message queue and advertises its ID. A task that wants to send or receive a message with this message queue first gets the message queue's ID. It then uses this ID to access the message queue.

For example, consider a typical server/client scenario where a server task **t1** (on CPU 1) reads requests from one message queue and replies to these requests with a different message queue. Task **t1** creates the request queue and advertises its ID by adding it to the name database assigning the name **requestQue**. If task **t2** (on CPU 0) wants to send a request to **t1**, it first gets the message queue ID by looking up the name **requestQue** in the name database. Before sending its first request, task **t2** creates a reply message queue. Instead of adding its ID to the database, it advertises the ID by sending it as part of the request message. When **t1** receives the request from the client, it finds in the message the ID of the queue to use when replying to that client. Task **t1** then sends the reply to the client by using this ID.

To pass messages between tasks on different CPUs, first create the message queue by calling **msgQSmCreate( )**. This routine returns a **MSG_Q_ID**. This ID is used for sending and receiving messages on the shared message queue.

Like their local counterparts, shared message queues can send both urgent or normal priority messages.

Figure 11-2 **Shared Message Queues**



Executes on CPU 2 after **task1**:

```
task2 ( )
   {
   ...
   msgQReceive (smMsgQId,...);
   ...
   }
```

Executes on CPU 1 before **task2**:

```
task1 ( )
   {
   ...
   msgQReceive (smMsgQId,...);
   ...
   }
```

The use of shared message queues and local message queues differs in several ways:

- The shared message queue task queueing order specified when a message queue is created must be FIFO. Figure 11-2 shows two tasks executing on different CPUs, both trying to receive a message from the same shared message queue. Task 1 executes first, and is put at the front of the queue because there are no messages in the message queue. Task 2 (executing on a different CPU) tries to receive a message from the message queue after task 1's attempt and is put on the queue behind task 1.
- Messages *cannot* be sent on a shared message queue at interrupt level. (This is true even in **NO_WAIT** mode.)

- Shared message queues cannot be deleted. Attempts to delete a shared message queue return **ERROR** and sets **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

To achieve optimum performance with shared message queues, align send and receive buffers on 4-byte boundaries.

To display the status of the shared message queue as well as a list of tasks pended on the queue, select **INCLUDE_MSG_Q_SHOW** for inclusion in the project facility VxWorks view and call **msgQShow( )**. The following example displays detailed information on the shared message queue 0x7f8c21 as indicated by the second argument (0 = summary display, 1 = detailed display).

```
-> msgQShow 0x7f8c21, 1
value = 0 = 0x0
```

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

```
Message Queue Id  : 0x7f8c21
Task Queuing      : FIFO
Message Byte Len  : 128
Messages Max      : 10
Messages Queued   : 0
Receivers Blocked : 1
Send timeouts     : 0
Receive timeouts  : 0
Receivers blocked :
TID           CPU Number           Shared TCB
---------- -------------------- --------------
0xd0618           1               0x1364204
```

Example 11-2 Shared Message Queues

In the following code example, two tasks executing on different CPUs use shared message queues to pass data to each other. The server task creates the request message queue, adds it to the name database, and reads a message from the queue. The client task gets the **smRequestQId** from the name database, creates a reply message queue, bundles the ID of the reply queue as part of the message, and sends the message to the server. The server gets the ID of the reply queue and uses it to send a message back to the client. This technique requires the use of the network byte-order conversion macros **htonl( )** and **ntohl( )**, because the numeric queue ID is passed over the network in a data field.

```
/* msgExample.h - shared message queue example header file */

#define MAX_MSG     (10)
#define MAX_MSG_LEN (100)
#define REQUEST_Q   "requestQue"
```

```
typedef struct message
    {
    MSG_Q_ID replyQId;
    char     clientRequest[MAX_MSG_LEN];
    } REQUEST_MSG;

/* server.c - shared message queue example server */

/* This file contains the code for the message queue server task. */

#include "vxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "stdio.h"
#include "smNameLib.h"
#include "msgExample.h"
#include "netinet/in.h"

#define REPLY_TEXT "Server received your request"

/*
 * serverTask - receive and process a request from a shared message queue
 */

STATUS serverTask (void)
    {
    MSG_Q_ID    smRequestQId;  /* request shared message queue */
    REQUEST_MSG request;       /* request text */

    /* create a shared message queue to handle requests */

    if ((smRequestQId = msgQSmCreate (MAX_MSG, sizeof (REQUEST_MSG),
        MSG_Q_FIFO)) == NULL)
        return (ERROR);

    /* add newly created request message queue to name database */

    if (smNameAdd (REQUEST_Q, smRequestQId, T_SM_MSG_Q) == ERROR)
        return (ERROR);

    /* read messages from request queue */

    FOREVER
        {
        if (msgQReceive (smRequestQId, (char *) &request, sizeof (REQUEST_MSG),
            WAIT_FOREVER) == ERROR)
            return (ERROR);

        /* process request - in this case simply print it */

        printf ("Server received the following message:\n%s\n",
            request.clientRequest);

        /* send a reply using ID specified in client's request message */
        if (msgQSend ((MSG_Q_ID) ntohl ((int) request.replyQId),
            REPLY_TEXT, sizeof (REPLY_TEXT),
            WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
            return (ERROR);
        }
    }

/* client.c - shared message queue example client */

/* This file contains the code for the message queue client task. */

#include "vxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "msgExample.h"
#include "netinet/in.h"

/*
 * clientTask - sends request to server and reads reply
 */

STATUS clientTask
    (
    char * pRequestToServer  /* request to send to the server */
                             /* limited to 100 chars */
    )
    {
    MSG_Q_ID    smRequestQId; /* request message queue */
    MSG_Q_ID  smReplyQId;   /* reply message queue */
    REQUEST_MSG request;      /* request text */
    int         objType;      /* dummy variable for smNameFind */
    char        serverReply[MAX_MSG_LEN]; /*buffer for server's reply */

    /* get request queue ID using its name */

    if (smNameFind (REQUEST_Q, (void **) &smRequestQId, &objType,
        WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* create reply queue, build request and send it to server */

    if ((smReplyQId = msgQSmCreate (MAX_MSG, MAX_MSG_LEN,
        MSG_Q_FIFO)) == NULL)
        return (ERROR);

    request.replyQId = (MSG_Q_ID) htonl ((int) smReplyQId);

    strcpy (request.clientRequest, pRequestToServer);

    if (msgQSend (smRequestQId, (char *) &request, sizeof (REQUEST_MSG),
        WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
```

```
/* read reply and print it */

if (msgQReceive (request.replyQId, serverReply, MAX_MSG_LEN,
    WAIT_FOREVER) == ERROR)
    return (ERROR);

printf ("Client received the following message:\n%s\n", serverReply);

return (OK);
}
```

11.2.4 Shared-Memory Allocator

The *shared-memory allocator* allows tasks on different CPUs to allocate and release variable size chunks of memory that are accessible from all CPUs with access to the shared-memory system. Two sets of routines are provided: low-level routines for manipulating user-created shared-memory partitions, and high-level routines for manipulating a shared-memory partition dedicated to the shared-memory system pool. (This organization is similar to that used by the local-memory manager, **memPartLib**.)

Shared-memory blocks can be allocated from different partitions. Both a shared-memory system partition and user-created partitions are available. User-created partitions can be created and used for allocating data blocks of a particular size. Memory fragmentation is avoided when fixed-sized blocks are allocated from user-created partitions dedicated to a particular block size.

Shared-Memory System Partition

To use the shared-memory system partition, a task allocates a shared-memory block and advertises its address. One way of advertising the ID is to add the address to the name database. The routine used to allocate a block from the shared-memory system partition returns a local address. Before the address is advertised to tasks on other CPUs, this local address must be converted to a global address. Any task that must use the shared memory must first get the address of the memory block and convert the global address to a local address. When the task has the address, it can use the memory.

However, to address issues of mutual exclusion, typically a shared semaphore is used to protect the data in the shared memory. Thus in a more common scenario, the task that creates the shared memory (and adds it to the database) also creates a shared semaphore. The shared semaphore ID is typically advertised by storing it in a field in the shared data structure residing in the shared-memory block. The first time a task must access the shared data structure, it looks up the address of the memory in the database and gets the semaphore ID from a field in the shared data structure. Whenever a task must access the shared data, it must first take the semaphore. Whenever a task is finished with the shared data, it must give the semaphore.

For example, assume two tasks executing on two different CPUs must share data. Task **t1** executing on CPU 1 allocates a memory block from the shared-memory system partition and converts the local address to a global address. It then adds the global address of the shared data to the name database with the name **mySharedData**. Task **t1** also creates a shared semaphore and stores the ID in the first field of the data structure residing in the shared memory. Task **t2** executing on CPU 2 looks up the name **mySharedData** in the name database to get the address of the shared memory. It then converts this address to a local address. Before accessing the data in the shared memory, **t2** gets the shared semaphore ID from the first field of the data structure residing in the shared-memory block. It then takes the semaphore before using the data and gives the semaphore when it is done using the data.

User-Created Partitions

To make use of user-created shared-memory partitions, a task creates a shared-memory partition and adds it to the name database. Before a task can use the shared-memory partition, it must first look in the name database to get the partition ID. When the task has the partition ID, it can access the memory in the shared-memory partition.

For example, task **t1** creates a shared-memory partition and adds it to the name database using the name **myMemPartition**. Task **t2** executing on another CPU wants to allocate memory from the new partition. Task **t2** first looks up **myMemPartition** in the name database to get the partition ID. It can then allocate memory from it, using the ID.

Using the Shared-Memory System Partition

The shared-memory system partition is analogous to the system partition for local memory. Table 11-4 lists routines for manipulating the shared-memory system partition.

Routines that return a pointer to allocated memory return a local address (that is, an address suitable for use from the local CPU). To share this memory across

Table 11-4 **Shared-Memory System Partition Routines**

Routine	Functionality
smMemMalloc( )	Allocates a block of shared system memory.
smMemCalloc( )	Allocates a block of shared system memory for an array.
smMemRealloc( )	Resizes a block of shared system memory.
smMemFree( )	Frees a block of shared system memory.
smMemShow( )	Displays usage statistics of the shared-memory system partition on the standard output device. This routine is automatically included if **INCLUDE_SM_OBJ** is selected for inclusion in the project facility VxWorks view.
smMemOptionsSet( )	Sets the debugging options for the shared-memory system partition.
smMemAddToPool( )	Adds memory to the shared-memory system pool.
smMemFindMax( )	Finds the size of the largest free block in the shared-memory system partition.

processors, this address must be converted to a global address before it is advertised to tasks on other CPUs. Before a task on another CPU uses the memory, it must convert the global address to a local address. Macros and routines are provided to convert between local addresses and global addresses; see the header file **smObjLib.h** and the reference entry for **smObjLib**.

Example 11-3 **Shared-Memory System Partition**

The following code example uses memory from the shared-memory system partition to share data between tasks on different CPUs. The first member of the data structure is a shared semaphore that is used for mutual exclusion. The send task creates and initializes the structure, then the receive task accesses the data and displays it.

```
/* buffProtocol.h - simple buffer exchange protocol header file */

#define BUFFER_SIZE   200           /* shared data buffer size */
#define BUFF_NAME     "myMemory"    /* name of data buffer in database */

typedef struct shared_buff
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;
```

```
/* buffSend.c - simple buffer exchange protocol send side */

/* This file writes to the shared memory. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*
 * buffSend - write to shared semaphore protected buffer
 */

STATUS buffSend (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;

    /* grab shared system memory */

    pSharedBuff = (SHARED_BUFF *) smMemMalloc (sizeof (SHARED_BUFF));

    /*
     * Initialize shared buffer structure before adding to database. The
     * protection semaphore is initially unavailable and the receiver blocks.
     */

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        return (ERROR);
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /*
     * Convert address of shared buffer to a global address and add to
     * database.
     */

    if (smNameAdd (BUFF_NAME, (void *) smObjLocalToGlobal (pSharedBuff),
                    T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* put data into shared buffer */

    sprintf (pSharedBuff->buff,"Hello from sender\n");

    /* allow receiver to read data by giving protection semaphore */

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

```
/* buffReceive.c - simple buffer exchange protocol receive side */

/* This file reads the shared memory. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*
 * buffReceive - receive shared semaphore protected buffer
 */

STATUS buffReceive (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;
    int           objType;

    /* get shared buffer address from name database */

    if (smNameFind (BUFF_NAME, (void **) &pSharedBuff,
                    &objType, WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* convert global address of buff to its local value */

    pSharedBuff = (SHARED_BUFF *) smObjGlobalToLocal (pSharedBuff);

    /* convert shared semaphore ID to host (local) byte order */

    mySemSmId = (SEM_ID) ntohl ((int) pSharedBuff->semSmId);

    /* take shared semaphore before reading the data buffer */

    if (semTake (mySemSmId,WAIT_FOREVER) != OK)
        return (ERROR);

    /* read data buffer and print it */

    printf ("Receiver reading from shared memory: %s\n", pSharedBuff->buff);

    /* give back the data buffer semaphore */

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

Using User-Created Partitions

Shared-memory partitions have a separate create routine, **memPartSmCreate( )**, that returns a **MEM_PART_ID**. After a user-defined shared-memory partition is created, routines in **memPartLib** operate on it transparently. Note that the address of the shared-memory area passed to **memPartSmCreate( )** (or **memPartAddToPool( )**) must be the global address.

Example 11-4 **User-Created Partition**

This example is similar to Example 11-3, which uses the shared-memory system partition. This example creates a user-defined partition and stores the shared data in this new partition. A shared semaphore is used to protect the data.

```
/* memPartExample.h - shared memory partition example header file */

#define CHUNK_SIZE      (2400)
#define MEM_PART_NAME   "myMemPart"
#define PART_BUFF_NAME  "myBuff"
#define BUFFER_SIZE     (40)

typedef struct shared_buff
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;

/* memPartSend.c - shared memory partition example send side */

/* This file writes to the user-defined shared memory partition. */

#include "vxWorks.h"
#include "memLib.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "smMemLib.h"
#include "stdio.h"
#include "memPartExample.h"

/*
 * memPartSend - send shared memory partition buffer
 */

STATUS memPartSend (void)
    {
    char *          pMem;
    PART_ID         smMemPartId;
    SEM_ID          mySemSmId;
    SHARED_BUFF *   pSharedBuff;
    /* allocate shared system memory to use for partition */

    pMem = smMemMalloc (CHUNK_SIZE);

    /* Create user defined partition using the previously allocated
     * block of memory.
     * WARNING: memPartSmCreate uses the global address of a memory
     * pool as first parameter.
     */

    if ((smMemPartId = memPartSmCreate (smObjLocalToGlobal (pMem), CHUNK_SIZE))
          == NULL)
        return (ERROR);

    /* allocate memory from partition */

    pSharedBuff = (SHARED_BUFF *) memPartAlloc ( smMemPartId,
                sizeof (SHARED_BUFF));
    if (pSharedBuff == 0)
        return (ERROR);

    /* initialize structure before adding to database */

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        return (ERROR);
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /* enter shared partition ID in name database */

    if (smNameAdd (MEM_PART_NAME, (void *) smMemPartId, T_SM_PART_ID) == ERROR)
        return (ERROR);

    /* convert shared buffer address to a global address and add to database */

    if (smNameAdd (PART_BUFF_NAME, (void *) smObjLocalToGlobal(pSharedBuff),
                   T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* send data using shared buffer */

    sprintf (pSharedBuff->buff,"Hello from sender\n");

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }

/* memPartReceive.c - shared memory partition example receive side */

/* This file reads from the user-defined shared memory partition. */

#include "vxWorks.h"
#include "memLib.h"
#include "stdio.h"
#include "semLib.h"
```

```
#include "semSmLib.h"
#include "stdio.h"
#include "memPartExample.h"

/*
 * memPartReceive - receive shared memory partition buffer
 *
 * execute on CPU 1 - use a shared semaphore to protect shared memory
 */

STATUS memPartReceive (void)
    {
    SHARED_BUFF * pBuff;
    SEM_ID        mySemSmId;
    int           objType;

    /* get shared buffer address from name database */

    if (smNameFind (PART_BUFF_NAME, (void **) &pBuff, &objType,
                    WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* convert global address of buffer to its local value */

    pBuff = (SHARED_BUFF *) smObjGlobalToLocal (pBuff);

    /* Grab shared semaphore before using the shared memory */

    mySemSmId = (SEM_ID) ntohl ((int) pBuff->semSmId);
    semTake (mySemSmId,WAIT_FOREVER);
    printf ("Receiver reading from shared memory: %s\n", pBuff->buff);
    semGive (mySemSmId);

    return (OK);
    }
```

Side Effects of Shared-Memory Partition Options

Like their local counterparts, shared-memory partitions (both system- and user-created) can have different options set for error handling; see the reference entries for **memPartOptionsSet( )** and **smMemOptionsSet( )**.

If the **MEM_BLOCK_CHECK** option is used in the following situation, the system can get into a state where the memory partition is no longer available. If a task attempts to free a bad block and a bus error occurs, the task is suspended. Because shared semaphores are used internally for mutual exclusion, the suspended task still has the semaphore, and no other task has access to the memory partition. By default, shared-memory partitions are created without the **MEM_BLOCK_CHECK** option.

11.3 Internal Considerations

11.3.1 System Requirements

The shared-memory region used by shared-memory objects must be visible to all CPUs in the system. Either dual-ported memory on the master CPU (CPU 0) or a separate memory board can be used. The shared-memory objects' anchor must be in the same address space as the shared-memory region. Note that the memory does *not* have to appear at the same local address for all CPUs.

⚠ **CAUTION:** Boards that make use of VxMP must support hardware test-and-set (indivisible read-modify-write cycle). PowerPC is an exception; see the current PowerPC architecture supplement.

All CPUs in the system must support indivisible read-modify-write cycle across the (VME) bus. The indivisible RMW is used by the spin-lock mechanism to gain exclusive access to internal shared data structures; see *11.3.2 Spin-lock Mechanism*, p. 452 for details. Because all the boards must support a hardware test-and-set, the constant **SM_TAS_TYPE** must be set to **SM_TAS_HARD** on the Parameters tab of the project facility VxWorks view.

CPUs must be notified of any event that affects them. The preferred method is for the CPU initiating the event to interrupt the affected CPU. The use of interrupts is dependent on the capabilities of the hardware. If interrupts cannot be used, a polling scheme can be employed, although this generally results in a significant performance penalty.

The maximum number of CPUs that can use shared-memory objects is 20 (CPUs numbered 0 through 19). The practical maximum is usually a smaller number that depends on the CPU, bus bandwidth, and application.

11.3.2 Spin-lock Mechanism

Internal shared-memory object data structures are protected against concurrent access by a *spin-lock mechanism*. The spin-lock mechanism is a loop where an attempt is made to gain exclusive access to a resource (in this case an internal data structure). An indivisible hardware read-modify-write cycle (hardware test-and-set) is used for this mutual exclusion. If the first attempt to take the lock fails, multiple attempts are made, each with a decreasing random delay between one attempt and the next. The average time it takes between the original attempt to take the lock and the first retry is 70 microseconds on an MC68030 at 20MHz.

Operating time for the spin-lock cycle varies greatly because it is affected by the processor cache, access time to shared memory, and bus traffic. If the lock is not obtained after the maximum number of tries specified by **SM_OBJ_MAX_TRIES** (defined in the Params tab of the properties window for shared memory objects in the VxWorks view), **errno** is set to **S_smObjLib_LOCK_TIMEOUT**. If this error occurs, set the maximum number of tries to a higher value. Note that any failure to take a spin-lock prevents proper functioning of shared-memory objects. In most cases, this is due to problems with the shared-memory configuration; see *11.5.2 Troubleshooting Techniques*, p.462.

11.3.3 Interrupt Latency

For the duration of the spin-lock, interrupts are disabled to avoid the possibility of a task being preempted while holding the spin-lock. As a result, the interrupt latency of each processor in the system is increased. However, the interrupt latency added by shared-memory objects is constant for a particular CPU.

11.3.4 Restrictions

Unlike local semaphores and message queues, shared-memory objects cannot be used at interrupt level. No routines that use shared-memory objects can be called from ISRs. An ISR is dedicated to handle time-critical processing associated with an external event; therefore, using shared-memory objects at interrupt time is not appropriate. On a multiprocessor system, run event-related time-critical processing on the CPU where the time-related interrupt occurred.

Note that shared-memory objects are allocated from dedicated shared-memory pools, and cannot be deleted.

When using shared-memory objects, the maximum number of each object type must be specified on the Params tab of the properties window; see *11.4.3 Initializing the Shared-Memory Objects Package*, p.456. If applications are creating more than the specified maximum number of objects, it is possible to run out of memory. If this happens, the shared object creation routine returns an error and **errno** is set to **S_memLib_NOT_ENOUGH_MEM**. To solve this problem, first increase the maximum number of shared-memory objects of corresponding type; see Table 11-5 for a list of the applicable configuration constants. This decreases the size of the shared-memory system pool because the shared-memory pool uses the remainder of the shared memory. If this is undesirable, increase both the number of the corresponding shared-memory objects and the size of the overall shared-memory region, **SM_OBJ_MEM_SIZE**. See *11.4 Configuration*, p.454 for a discussion of the constants used for configuration.

11.3.5 Cache Coherency

When dual-ported memory is used on some boards without MMU or bus snooping mechanisms, the data cache must be disabled for the shared-memory region on the master CPU. If you see the following error message, make sure that the constant **INCLUDE_CACHE_ENABLE** is not selected for inclusion in the VxWorks view:

```
usrSmObjInit - cache coherent buffer not available. Giving up.
```

11.4 Configuration

To include shared-memory objects in VxWorks, select **INCLUDE_SM_OBJ** for inclusion in the project facility VxWorks view. Most of the configuration is already done automatically from **usrSmObjInit( )** in **usrConfig.c**. However, you may also need to modify some values in the Params tab of the properties window to reflect your configuration; these are described in this section.

11.4.1 Shared-Memory Objects and Shared-Memory Network Driver

Shared-memory objects and the shared-memory network[2] use the same memory region, anchor address, and interrupt mechanism. Configuring the system to use shared-memory objects is similar to configuring the shared-memory network driver. For a more detailed description of configuring and using the shared-memory network, see *VxWorks Network Programmer's Guide: Data Link Layer Network Components*. If the default value for the shared-memory anchor address is modified, the anchor must be on a 256-byte boundary.

One of the most important aspects of configuring shared-memory objects is computing the address of the shared-memory anchor. The shared-memory anchor is a location accessible to all CPUs on the system, and is used by both VxMP and

[2] Also known as the *backplane network*.

the shared-memory network driver. The anchor stores a pointer to the shared-memory header, a pointer to the shared-memory packet header (used by the shared-memory network driver), and a pointer to the shared-memory object header.

The address of the anchor is defined in the Params tab of the Properties window with the constant **SM_ANCHOR_ADRS**. If the processor is booted with the shared-memory network driver, the anchor address is the same value as the boot device (**sm=***anchorAddress*). The shared-memory object initialization code uses the value from the boot line instead of the constant. If the shared-memory network driver is not used, modify the definition of **SM_ANCHOR_ADRS** as appropriate to reflect your system.

Two types of interrupts are supported and defined by **SM_INT_TYPE**: mailbox interrupts and bus interrupts (see *VxWorks Network Programmer's Guide: Data Link Layer Network Components*). Mailbox interrupts (**SM_INT_MAILBOX**) are the preferred method, and bus interrupts (**SM_INT_BUS**) are the second choice. If interrupts cannot be used, a polling scheme can be employed (**SM_INT_NONE**), but this is much less efficient.

When a CPU initializes its shared-memory objects, it defines the interrupt type as well as three interrupt arguments. These describe how the CPU is notified of events. These values can be obtained for any attached CPU by calling **smCpuInfoGet( )**.

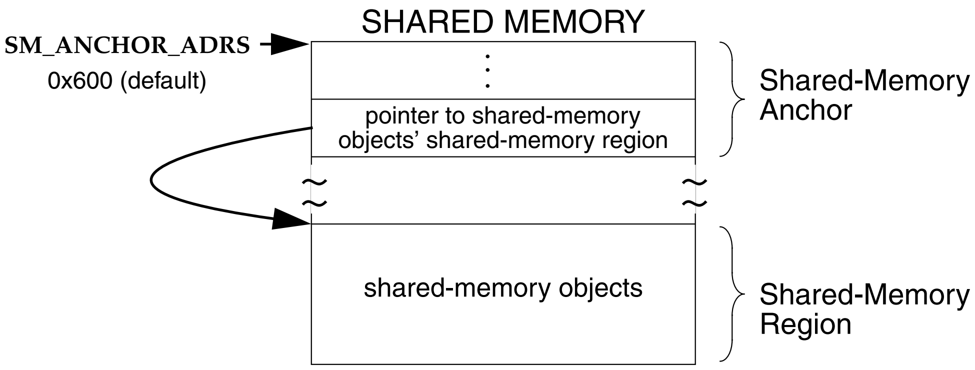
The default interrupt method for a target is defined by **SM_INT_TYPE**, **SM_INT_ARG1**, **SM_INT_ARG2**, and **SM_INT_ARG3** on the Params tab.

11.4.2 Shared-Memory Region

Shared-memory objects rely on a shared-memory region that is visible to all processors. This region is used to store internal shared-memory object data structures and the shared-memory system partition.

The shared-memory region is usually in dual-ported RAM on the master, but it can also be located on a separate memory card. The shared-memory region address is defined when configuring the system as an offset from the shared-memory anchor address, **SM_ANCHOR_ADRS**, as shown in Figure 11-3.

Figure 11-3 **Shared-Memory Layout**



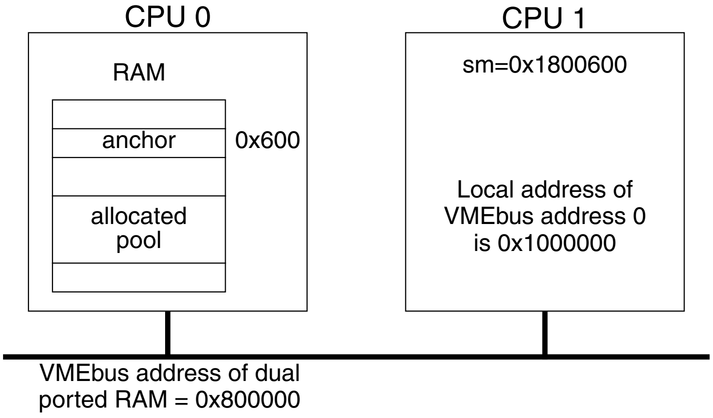
11.4.3 Initializing the Shared-Memory Objects Package

Shared-memory objects are initialized by default in the routine **usrSmObjInit( )** in **targetsrc/config/usrSmObj.c**. The configuration steps taken for the master CPU differ slightly from those taken for the slaves.

The address for the shared-memory pool must be defined. If the memory is off-board, the value must be calculated (see Figure 11-5).

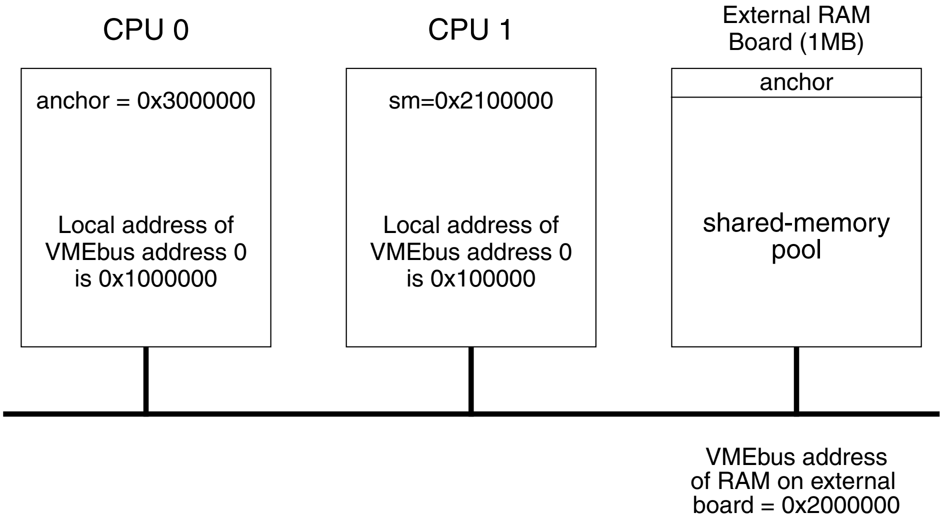
The example configuration in Figure 11-4 uses the shared memory in the master CPU's dual-ported RAM.

Figure 11-4 **Example Configuration: Dual-Ported Memory**


CPU 0
RAM
anchor
0x600
allocated pool
CPU 1
sm=0x1800600
Local address of VMEbus address 0 is 0x1000000
VMEbus address of dual ported RAM = 0x800000

On the Params tab of the properties window for the master, **SM_OFF_BOARD** is FALSE and **SM_ANCHOR_ADRS** is 0x600. **SM_OBJ_MEM_ADRS** is set to **NONE**, because on-board memory is used (it is malloc'ed at run-time); **SM_OBJ_MEM_SIZE** is set to 0x20000. For the slave, the board maps the base of the VME bus to the address 0x1000000. **SM_OFF_BOARD** is TRUE and the anchor address is 0x1800600. This is calculated by taking the VMEbus address (0x800000) and adding it to the anchor address (0x600). Many boards require further address translation, depending on where the board maps VME memory. In this example, the anchor address for the slave is 0x1800600, because the board maps the base of the VME bus to the address 0x1000000.

Figure 11-5 **Example Configuration: an External Memory Board**


CPU 0
CPU 1
External RAM Board (1MB)
anchor = 0x3000000
Local address of VMEbus address 0 is 0x1000000
sm=0x2100000
Local address of VMEbus address 0 is 0x100000
anchor
shared-memory pool
VMEbus address of RAM on external board = 0x2000000

In the example configuration in Figure 11-5, the shared memory is on a separate memory board. On the Params tab for the master, **SM_OFF_BOARD** is TRUE, **SM_ANCHOR_ADRS** is 0x3000000, **SM_OBJ_MEM_ADRS** is set to **SM_ANCHOR_ADRS**, and **SM_OBJ_MEM_SIZE** is set to 0x100000. For the slave board, **SM_OFF_BOARD** is TRUE and the anchor address is 0x2100000. This is calculated by taking the VMEbus address of the memory board (0x2000000) and adding it to the local VMEbus address (0x100000).

Some additional configuration is sometimes required to make the shared memory non-cacheable, because the shared-memory pool is accessed by all processors on the backplane. By default, boards with an MMU have the MMU turned on. With the MMU on, memory that is off-board must be made non-cacheable. This is done using the data structure **sysPhysMemDesc** in **sysLib.c**. This data structure must contain a virtual-to-physical mapping for the VME address space used for the shared-memory pool, and mark the memory as non-cacheable. (Most BSPs include this mapping by default.) See *12.3 Virtual Memory Configuration*, p.466, for additional information.

⚠ **CAUTION:** For the MC68K in general, if the MMU is off, data caching must be turned off globally; see the reference entry for **cacheLib**.

When shared-memory objects are initialized, the memory size as well as the maximum number of each object type must be specified. The master processor specifies the size of memory using the constant **SM_OBJ_MEM_SIZE**. Symbolic constants are used to set the maximum number of different objects. These constants are specified on the Params tab of the properties window. See Table 11-5 for a list of these constants.

Table 11-5 **Configuration Constants for Shared-Memory Objects**

Symbolic Constant	Default Value	Description
SM_OBJ_MAX_TASK	40	Maximum number of tasks using shared-memory objects.
SM_OBJ_MAX_SEM	30	Maximum number of shared semaphores (counting and binary).
SM_OBJ_MAX_NAME	100	Maximum number of names in the name database.
SM_OBJ_MAX_MSG_Q	10	Maximum number of shared message queues.
SM_OBJ_MAX_MEM_PART	4	Maximum number of user-created shared-memory partitions.

If the size of the objects created exceeds the shared-memory region, an error message is displayed on CPU 0 during initialization. After shared memory is configured for the shared objects, the remainder of shared memory is used for the shared-memory system partition.

The routine **smObjShow( )** displays the current number of used shared-memory objects and other statistics, as follows:

```
-> smObjShow
value = 0 = 0x0
```

The **smObjShow( )** routine is automatically included if **INCLUDE_SM_OBJ** is selected for inclusion in the project facility VxWorks view. The output of **smObjShow( )** is sent to the standard output device, and looks like the following:

```
Shared Mem Anchor Local Addr : 0x600
Shared Mem Hdr Local Addr    : 0x363ed0
Attached CPU                 : 2
Max Tries to Take Lock       : 0
Shared Object Type      Current      Maximum      Available
------------------      -------      -------      ---------
Tasks                         1           40             39
Binary Semaphores             3           30             27
Counting Semaphores           0           30             27
Messages Queues               1           10              9
Memory Partitions             1            4              3
Names in Database             5          100             95
```

⚠ **CAUTION:** If the master CPU is rebooted, it is necessary to reboot all the slaves. If a slave CPU is to be rebooted, it must not have tasks pended on a shared-memory object.

11.4.4 Configuration Example

The following example shows the configuration for a multiprocessor system with three CPUs. The master is CPU 0, and shared memory is configured from its dual-ported memory. This application has 20 tasks using shared-memory objects, and uses 12 message queues and 20 semaphores. The maximum size of the name database is the default value (100), and only one user-defined memory partition is required. On CPU 0, the shared-memory pool is configured to be on-board. This memory is allocated from the processor's system memory. On CPU 1 and CPU 2, the shared-memory pool is configured to be off-board. Table 11-6 shows the values set on the Params tab of the properties window for **INCLUDE_SM_OBJECTS** in the project facility.

Note that for the slave CPUs, the value of **SM_OBJ_MEM_SIZE** is not actually used.

Table 11-6 **Configuration Settings for Three CPU System**

CPU	Symbolic Constant	Value
Master (**CPU 0**)	**SM_OBJ_MAX_TASK**	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	**FALSE**
	SM_MEM_ADRS	NONE
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000
Slaves (**CPU 1**, **CPU 2**)	**SM_OBJ_MAX_TASK**	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	**FALSE**
	SM_ANCHOR_ADRS	(char *) 0xfb800000
	SM_MEM_ADRS	**SM_ANCHOR_ADRS**
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000

11.4.5 Initialization Steps

Initialization is performed by default in **usrSmObjInit( )**, in **targetsrc/config/usrSmObj.c**. On the master CPU, the initialization of shared-memory objects consists of the following:

1. Setting up the shared-memory objects header and its pointer in the shared-memory anchor, with **smObjSetup( )**.
2. Initializing shared-memory object parameters for this CPU, with **smObjInit( )**.
3. Attaching the CPU to the shared-memory object facility, with **smObjAttach( )**.

On slave CPUs, only steps 2 and 3 are required.

The routine **smObjAttach( )** checks the setup of shared-memory objects. It looks for the *shared-memory heartbeat* to verify that the facility is running. The shared-memory heartbeat is an unsigned integer that is incremented once per second by the master CPU. It indicates to the slaves that shared-memory objects are initialized, and can be used for debugging. The heartbeat is the first field in the shared-memory object header; see *11.5 Troubleshooting*, p. 461.

11.5 Troubleshooting

Problems with shared-memory objects can be due to a number of causes. This section discusses the most common problems and a number of troubleshooting tools. Often, you can locate the problem by rechecking your hardware and software configurations.

11.5.1 Configuration Problems

Use the following list to confirm that your system is properly configured:

- Be sure to verify that the constant **INCLUDE_SM_OBJ** is selected for inclusion in the project facility VxWorks view for each processor using VxMP.
- Be sure the anchor address specified is the address seen by the CPU. This can be defined with the constant **SM_ANCHOR_ADRS** in the Params tab of the properties window or at boot time (**sm=**) if the target is booted with the shared-memory network.

- If there is heavy bus traffic relating to shared-memory objects, bus errors can occur. Avoid this problem by changing the bus arbitration mode or by changing relative CPU priorities on the bus.
- If **memAddToPool( )**, **memPartSmCreate( )**, or **smMemAddToPool( )** fail, check that any address you are passing to these routines is in fact a global address.

11.5.2 Troubleshooting Techniques

Use the following techniques to troubleshoot any problems you encounter:

- The routine **smObjTimeoutLogEnable( )** enables or disables the printing of an error message indicating that the maximum number of attempts to take a spin-lock has been reached. By default, message printing is enabled.
- The routine **smObjShow( )** displays the status of the shared-memory objects facility on the standard output device. It displays the maximum number of tries a task took to get a spin-lock on a particular CPU. A high value can indicate that an application might run into problems due to contention for shared-memory resources.
- The shared-memory heartbeat can be checked to verify that the master CPU has initialized shared-memory objects. The shared-memory heartbeat is in the first 4-byte word of the shared-memory object header. The offset to the header is in the sixth 4-byte word in the shared-memory anchor. (See *VxWorks Network Programmer's Guide: Data Link Layer Network Components*.)

  Thus, if the shared-memory anchor were located at 0x800000:

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

  The offset to the shared-memory object header is 0x170. To view the shared-memory object header display 0x800170:

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

  In the preceding example, the value of the shared-memory heartbeat is 0x50. Display this location again to ensure that the heartbeat is alive; if its value has changed, shared-memory objects are initialized.

- The global variable **smIfVerbose**, when set to 1 (TRUE), causes shared-memory interface error messages to print to the console, along with additional details of shared-memory operations. This variable enables you to get run-time information from the device driver level that would be unavailable at the debugger level. The default setting for **smIfVerbose** is 0 (FALSE). That can be reset programmatically or from the shell.

12

Virtual Memory Interface

Basic Support and Optional Component VxVMI

12.1 Introduction

VxWorks provides two levels of virtual memory support. The basic level is bundled with VxWorks and provides caching on a per-page basis. The full level is unbundled, and requires the optional component VxVMI. VxVMI provides write protection of text segments and the VxWorks exception vector table, and an architecture-independent interface to the CPU's memory management unit (MMU). For information on how to install VxVMI, see the *Tornado Getting Started Guide*.

This chapter contains the following sections:

- A description of the basic level of support.
- Configuration guidelines applicable to both levels of support.
- Two sections that apply only to the optional component VxVMI:
  - One for general use, discussing the write protection implemented by VxVMI.
  - One that describes a set of routines for manipulating the MMU. VxVMI provides low-level routines for interfacing with the MMU in an architecture-independent manner, allowing you to implement your own virtual memory systems.

12.2 Basic Virtual Memory Support

For systems with an MMU, VxWorks allows you to perform DMA and interprocessor communication more efficiently by rendering related buffers noncacheable. This is necessary to ensure that data is not being buffered locally when other processors or DMA devices are accessing the same memory location. Without the ability to make portions of memory noncacheable, caching must be turned off globally (resulting in performance degradation) or buffers must be flushed/invalidated manually.

Basic virtual memory support is included by selecting **INCLUDE_MMU_BASIC** in the project facility VxWorks view; see *12.3 Virtual Memory Configuration*, p. 466. It is also possible to allocate noncacheable buffers using **cacheDmaMalloc( )**; see the reference entry for **cacheLib**.

12.3 Virtual Memory Configuration

The following discussion of configuration applies to both bundled and unbundled virtual memory support. In the project facility, define the constants in Table 12-1 to reflect your system configuration.

Table 12-1 **MMU Configuration Constants**

Constant	Description
INCLUDE_MMU_BASIC	Basic MMU support without VxVMI option.
INCLUDE_MMU_FULL	Full MMU support with the VxVMI option.
INCLUDE_PROTECT_TEXT	Text segment protection (requires full MMU support).
INCLUDE_PROTECT_VEC_TABLE	Exception vector table protection (requires full MMU support).

The appropriate default page size for your processor (4 KB or 8KB) is defined by **VM_PAGE_SIZE** in your BSP. If you must change this value for some reason, redefine **VM_PAGE_SIZE** in **config.h**. (See the *Tornado User's Guide: Configuration and Build*.)

To make memory noncacheable, it must have a virtual-to-physical mapping. The data structure **PHYS_MEM_DESC** in **vmLib.h** defines the parameters used for mapping physical memory. Each board's memory map is defined in **sysLib.c** using **sysPhysMemDesc** (which is declared as an array of **PHYS_MEM_DESC**). In addition to defining the initial state of the memory pages, the **sysPhysMemDesc** structure defines the virtual addresses used for mapping virtual-to-physical memory. For a discussion of page states, see *Page States*, p. 470.

Modify the **sysPhysMemDesc** structure to reflect your system configuration. For example, you may need to add the addresses of interprocessor communication buffers not already included in the structure. Or, you may need to map and make noncacheable the VMEbus addresses of the shared-memory data structures. Most board support packages have a section of VME space defined in **sysPhysMemDesc**; however, this may not include all the space required by your system configuration.

I/O devices and memory not already included in the structure must also be mapped and made noncacheable. In general, off-board memory regions are specified as noncacheable; see *VxWorks Network Programmer's Guide: Data Link Layer Network Components*.

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 **VM_PAGE_SIZE**. Specifying elements of **sysPhysMemDesc** that are not page-aligned leads to crashes during VxWorks initialization.

The following example configuration consists of multiple CPUs using the shared-memory network. A separate memory board is used for the shared-memory pool. Because this memory is not already mapped, it must be added to **sysPhysMemDesc** for all the boards on the network. The memory starts at 0x4000000 and must be made noncacheable, as shown in the following code excerpt:

```
/* shared memory */
{
(void *) 0x4000000,         /* virtual address */
(void *) 0x4000000,         /* physical address */
0x20000,              /* length */
/* initial state mask */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |VM_STATE_MASK_CACHEABLE,
/* initial state */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

For MC680*x*0 boards, the virtual address *must* be the same as the physical address. For other boards, the virtual and physical addresses are the same as a matter of convention.

12.4 General Use

This section describes VxVMI's general use and configuration for write-protecting text segments and the exception vector table.

VxVMI uses the MMU to prevent portions of memory from being overwritten. This is done by write-protecting pages of memory. Not all target hardware supports write protection; see the architecture appendices in this manual for further information. For most architectures, the page size is 8KB. An attempt to write to a memory location that is write-protected causes a bus error.

When VxWorks is loaded, all text segments are write-protected; see *12.3 Virtual Memory Configuration*, p.466. The text segments of additional object modules loaded using **ld( )** are automatically marked as read-only. When object modules are loaded, memory to be write-protected is allocated in page-size increments. No additional steps are required to write-protect application code.

During system initialization, VxWorks write-protects the exception vector table. The only way to modify the interrupt vector table is to use the routine **intConnect( )**, which write-enables the exception vector table for the duration of the call.

To include write protection, select the following in the project facility VxWorks view:

INCLUDE_MMU_FULL
INCLUDE_PROTECT_TEXT
INCLUDE_PROTECT_VEC_TABLE

12.5 Using the MMU Programmatically

This section describes the facilities provided for manipulating the MMU programmatically using low-level routines in **vmLib**. You can make data private to a task or code segment, make portions of memory noncacheable, or write-protect portions of memory. The fundamental structure used to implement virtual memory is the *virtual memory context* (VMC).

For a summary of the VxVMI routines, see the reference entry for **vmLib**.

12.5.1 Virtual Memory Contexts

A virtual memory context (**VM_CONTEXT**, defined in **vmLib**) is made up of a translation table and other information used for mapping a virtual address to a physical address. Multiple virtual memory contexts can be created and swapped in and out as desired.

Global Virtual Memory

Some system objects, such as text segments and semaphores, must be accessible to all tasks in the system regardless of which virtual memory context is made current. These objects are made accessible by means of *global virtual memory*. Global virtual memory is created by mapping all the physical memory in the system (the mapping is defined in **sysPhysMemDesc**) to the identical address in the virtual memory space. In the default system configuration, this initially gives a one-to-one relationship between physical memory and global virtual memory; for example, virtual address 0x5000 maps to physical address 0x5000. On some architectures, it is possible to use **sysPhysMemDesc** to set up virtual memory so that the mapping of virtual-to-physical addresses is not one-to-one; see *12.3 Virtual Memory Configuration*, p.466 for additional information.

Global virtual memory is accessible from all virtual memory contexts. Modifications made to the global mapping in one virtual memory context appear in all virtual memory contexts. Before virtual memory contexts are created, add all global memory with **vmGlobalMap( )**. Global memory that is added after virtual memory contexts are created may not be available to existing contexts.

Initialization

Global virtual memory is initialized by **vmGlobalMapInit( )** in **usrMmuInit( )**, which is called from **usrRoot( )**. The routine **usrMmuInit( )** is in *installDir***/target/src/config/usrMmuInit.c**, and creates global virtual memory using **sysPhysMemDesc**. It then creates a default virtual memory context and makes the default context current. Optionally, it also enables the MMU.

Page States

Each virtual memory page (typically 8KB) has a state associated with it. A page can be valid/invalid, writable/nonwritable, or cacheable/noncacheable. See Table 12-2 for the associated constants.

Table 12-2 **State Flags**

Constant	Description
VM_STATE_VALID	Valid translation
VM_STATE_VALID_NOT	Invalid translation
VM_STATE_WRITABLE	Writable memory
VM_STATE_WRITABLE_NOT	Read-only memory
VM_STATE_CACHEABLE	Cacheable memory
VM_STATE_CACHEABLE_NOT	Noncacheable memory

Validity

A valid state indicates the virtual-to-physical translation is true. When the translation tables are initialized, global virtual memory is marked as valid. All other virtual memory is initialized as invalid.

Writability

Pages can be made read-only by setting the state to nonwritable. This is used by VxWorks to write-protect all text segments.

Cacheability

The caching of memory pages can be prevented by setting the state flags to noncacheable. This is useful for memory that is shared between processors (including DMA devices).

Change the state of a page with the routine **vmStateSet( )**. In addition to specifying the state flags, a state mask must describe which flags are being changed; see Table 12-3. Additional architecture-dependent states are specified in **vmLib.h**.

Table 12-3 **State Masks**

Constant	Description
VM_STATE_MASK_VALID	Modifies valid flag.
VM_STATE_MASK_WRITABLE	Modifies write flag.
VM_STATE_MASK_CACHEABLE	Modifies cache flag.

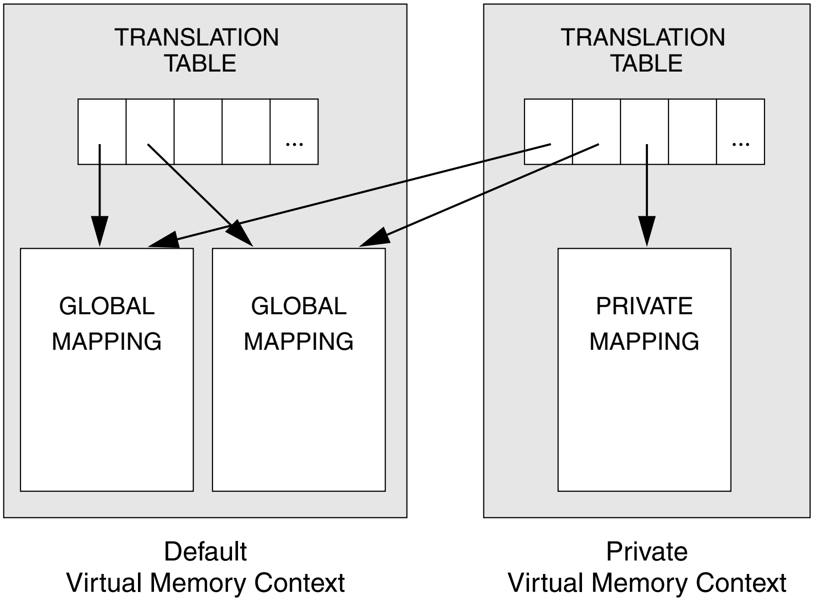
12.5.2 Private Virtual Memory

Private virtual memory can be created by creating a new virtual memory context. This is useful for protecting data by making it inaccessible to other tasks or by limiting access to specific routines. Virtual memory contexts are not automatically created for tasks, but can be created and swapped in and out in an application-specific manner.

At system initialization, a default context is created. All tasks use this default context. To create private virtual memory, a task must create a new virtual memory context using **vmContextCreate( )**, and make it current. All virtual memory contexts share the global mappings that are created at system initialization; see Figure 12-1. Only the valid virtual memory in the current virtual memory context (including global virtual memory) is accessible. Virtual memory defined in other virtual memory contexts is not accessible. To make another memory context current, use **vmCurrentSet( )**.

To create a new virtual-to-physical mapping, use **vmMap( )**; both the physical and virtual address must be determined in advance. The physical memory (which must be page aligned) can be obtained using **valloc( )**. The easiest way to determine the virtual address is to use **vmGlobalInfoGet( )** to find a virtual page that is not a global mapping. With this scheme, if multiple mappings are required, a task must keep track of its own private virtual memory pages to guarantee it does not map the same non-global address twice.

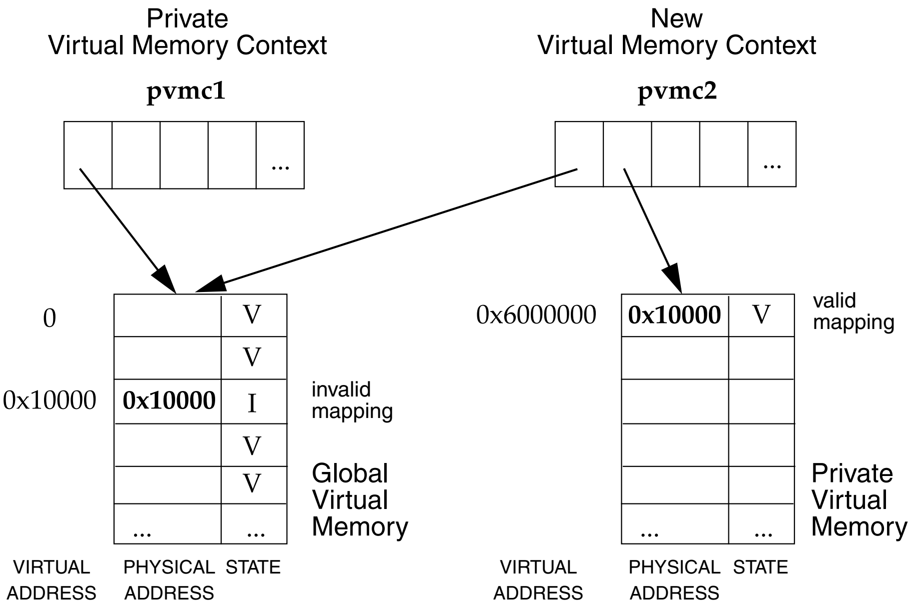
Figure 12-1 **Global Mappings of Virtual Memory**



When physical pages are mapped into new sections of the virtual space, the physical page is accessible from two different virtual addresses (a condition known as *aliasing*): the newly mapped virtual address and the virtual address equal to the physical address in the global virtual memory. This can cause problems for some architectures, because the cache may hold two different values for the same underlying memory location. To avoid this, invalidate the virtual page (using **vmStateSet( )**) in the global virtual memory. This also ensures that the data is accessible only when the virtual memory context containing the new mapping is current.

Figure 12-2 depicts two private virtual memory contexts. The new context (**pvmc2**) maps virtual address 0x6000000 to physical address 0x10000. To prevent access to this address from outside of this virtual context (**pvmc1**), the corresponding physical address (0x10000) must be set to invalid. If access to the memory is made using address 0x10000, a bus error occurs because that address is now invalid.

Figure 12-2 **Mapping Private Virtual Memory**



Example 12-1 **Private Virtual Memory Contexts**

In the following code example, private virtual memory contexts are used for allocating memory from a task's private memory partition. The setup routine, **contextSetup( )**, creates a private virtual memory context that is made current during a context switch. The virtual memory context is stored in the field **spare1** in the task's TCB. Switch hooks are used to save the old context and install the task's private context. Note that the use of switch hooks increases the context switch time. A user-defined memory partition is created using the private virtual memory context. The partition ID is stored in **spare2** in the tasks TCB. Any task wanting a private virtual memory context must call **contextSetup( )**. A sample task to test the code is included.

```
/* contextExample.h - header file for vm contexts used by switch hooks */

#define NUM_PAGES (3)

/* context.c - use context switch hooks to make task private context current */

#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
```

```
#include "taskLib.h"
#include "taskHookLib.h"
#include "memLib.h"
#include "contextExample.h"

void privContextSwitch (WIND_TCB *pOldTask, WIND_TCB *pNewTask);

/*
 * initContextSetup - install context switch hook
 */

STATUS initContextSetup ( )
    {
    /* Install switch hook */

    if (taskSwitchHookAdd ((FUNCPTR) privContextSwitch) == ERROR)
        return (ERROR);

    return (OK);
    }

/*
 * contextSetup - initialize context and create separate memory partition
 *
 * Call only once for each task that wants a private context.
 *
 * This could be made into a create-hook routine if every task on the
 * system needs a private context. To use as a create hook, the code for
 * installing the new virtual memory context should be replaced by simply
 * saving the new context in spare1 of the task's TCB.
 */

STATUS contextSetup (void)
    {
    VM_CONTEXT_ID pNewContext;
    int pageSize;
    int pageBlkSize;
    char * pPhysAddr;
    char * pVirtAddr;
    UINT8 * globalPgBlkArray;
    int newMemSize;
    int index;
    WIND_TCB * pTcb;

    /* create context */

    pNewContext = vmContextCreate();

    /* get page and page block size */

    pageSize = vmPageSizeGet ();
    pageBlkSize = vmPageBlockSizeGet ();
    newMemSize = pageSize * NUM_PAGES;

    /* allocate physical memory that is page aligned */
    if ((pPhysAddr = (char *) valloc (newMemSize)) == NULL)
        return (ERROR);

    /* Select virtual address to map. For this example, since only one page
     * block is used per task, simply use the first address that is not a
     * global mapping. vmGlobalInfoGet( ) returns a boolean array where each
     * element corresponds to a block of virtual memory.
     */

    globalPgBlkArray = vmGlobalInfoGet();
    for (index = 0; globalPgBlkArray[index] == TRUE; index++)
        ;
    pVirtAddr = (char *) (index * pageBlkSize);

    /* map physical memory to new context */

    if (vmMap (pNewContext, pVirtAddr, pPhysAddr, newMemSize) == ERROR)
        {
        free (pPhysAddr);
        return (ERROR);
        }

    /*
     * Set state in global virtual memory to be invalid - any access to
     * this memory must be done through new context.
     */

    if (vmStateSet(pNewContext, pPhysAddr, newMemSize, VM_STATE_MASK_VALID,
                    VM_STATE_VALID_NOT) == ERROR)
        return (ERROR);

    /* get tasks TCB */

    pTcb = taskTcb (taskIdSelf());

    /* change virtual memory contexts */

    /*
     * Stash the current vm context in the spare TCB field -- the switch
     * hook will install this when this task gets swapped out.
     */

    pTcb->spare1 = (int) vmCurrentGet();

    /* install new tasks context */

    vmCurrentSet (pNewContext);

    /* create new memory partition and store id in task's TCB */

    if ((pTcb->spare2 = (int) memPartCreate (pVirtAddr,newMemSize)) == NULL)
        return (ERROR);

    return (OK);
    }
```

```
/*
 * privContextSwitch - routine to be executed on a context switch
 *
 * If old task had private context, save it. If new task has private
 * context, install it.
 */

void privContextSwitch
    (
    WIND_TCB *pOldTcb,
    WIND_TCB *pNewTcb
    )

    {
    VM_CONTEXT_ID pContext = NULL;

    /* If previous task had private context, save it--reset previous context. */

    if (pOldTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pOldTcb->spare1;
        pOldTcb->spare1 = (int) vmCurrentGet ();

        /* restore old context */

        vmCurrentSet (pContext);
        }

    /*
     * If next task has private context, map new context and save previous
     * context in task's TCB.
     */

    if (pNewTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pNewTcb->spare1;
    pNewTcb->spare1 = (int) vmCurrentGet();

        /* install new tasks context */

        vmCurrentSet (pContext);
        }
    }

/* taskExample.h - header file for testing VM contexts used by switch hook */

/* This code is used by the sample task. */

#define MAX (10000000)

typedef struct myStuff {
    int stuff;
    int myStuff;
    } MY_DATA;
```

```
/* testTask.c - task code to test switch hooks */

#include "vxWorks.h"
#include "memLib.h"
#include "taskLib.h"
#include "stdio.h"
#include "vmLib.h"
#include "taskExample.h"

IMPORT char *string = "test\n";

MY_DATA *pMem;

/*
 * testTask - allocate private memory and use it
 *
 * Loop forever, modifying memory and printing out a global string. Use this
 * in conjunction with testing from the shell. Since pMem points to private
 * memory, the shell should generate a bus error when it tries to read it.
 * For example:
 *     -> sp testTask
 *     -> d pMem
 */

STATUS testTask (void)
    {
    int val;
    WIND_TCB *myTcb;

    /* install private context */

    if (contextSetup () == ERROR)
        return (ERROR);

    /* get TCB */

    myTcb = taskTcb (taskIdSelf ());

    /* allocate private memory */

    if ((pMem = (MY_DATA *) memPartAlloc((PART_ID) myTcb->spare2,
         sizeof (MY_DATA))) == NULL)
        return (ERROR);

    /*
     * Forever, modify data in private memory and display string in
     * global memory.
     */

    FOREVER
    {
        for (val = 0; val <= MAX; val++)
            {
            /* modify structure */
            pMem->stuff = val;
            pMem->myStuff = val / 2;

            /* make sure can access global virtual memory */

            printf (string);

            taskDelay (sysClkRateGet() * 10);
            }
        }
    return (OK);
    }

/*
 * testVmContextGet - return a task's virtual memory context stored in TCB
 *
 * Used with vmContextShow()[1] to display a task's virtual memory context.
 * For example, from the shell, type:
 *   -> tid = sp (testTask)
 *   -> vmContextShow (testVmContextGet (tid))
 */

VM_CONTEXT_ID testVmContextGet
    (
    UINT tid
    )
    {
    return ((VM_CONTEXT_ID) ((taskTcb (tid))->spare1));
    }
```

12.5.3 Noncacheable Memory

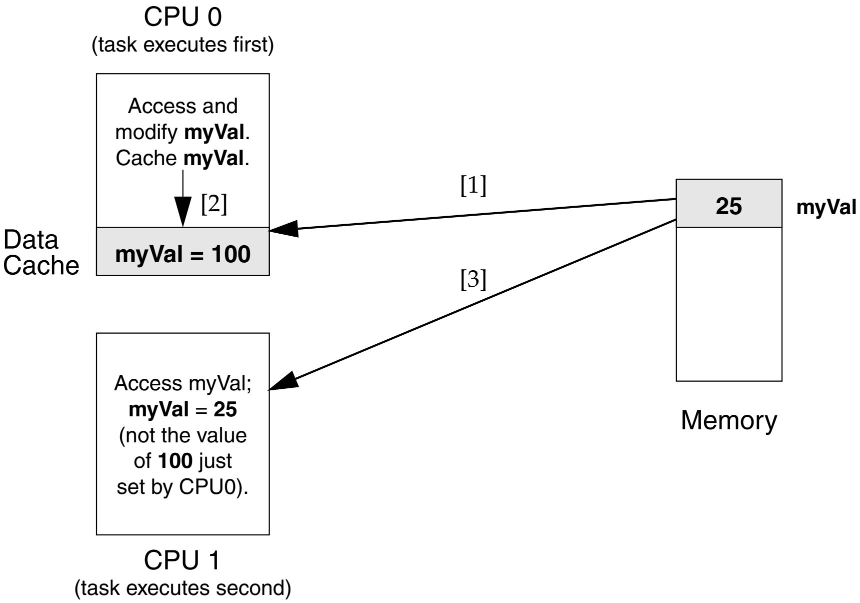
Architectures that do not support bus snooping must disable the memory caching that is used for interprocessor communication (or by DMA devices). If multiple processors are reading from and writing to a memory location, you must guarantee that when the CPU accesses the data, it is using the most recent value. If caching is used in one or more CPUs in the system, there can be a local copy of the data in one of the CPUs' data caches.

In the example in Figure 12-3, a system with multiple CPUs share data, and one CPU on the system (CPU 0) caches the shared data. A task on CPU 0 reads the data [1] and then modifies the value [2]; however, the new value may still be in the cache and not flushed to memory when a task on another CPU (CPU 1) accesses it [3].

[1] This routine is *not* built in to the Tornado shell. To use it from the Tornado shell, you must define **INCLUDE_MMU_FULL_SHOW** in your VxWorks configuration; see the *Tornado User's Guide: Projects*. When invoked this routine's output is sent to the standard output device.

Thus the value of the data used by the task on CPU 1 is the old value and does not reflect the modifications done by the task on CPU 0; that value is still in CPU 0's data cache [2].

Figure 12-3 **Example of Possible Problems with Data Caching**


CPU 0
(task executes first)
Access and modify **myVal**. Cache **myVal**.
[2]
[1]
25
myVal
Data Cache
myVal = 100
[3]
Access myVal; **myVal** = **25** (not the value of **100** just set by CPU0).
Memory
CPU 1
(task executes second)

To disable caching on a page basis, use **vmStateSet( )**; for example:

```
vmStateSet (pContext, pSData, len, VM_STATE_MASK_CACHEABLE, VM_STATE_CACHEABLE_NOT)
```

To allocate noncacheable memory, see the reference entry for **cacheDmaMalloc( )**.

12.5.4 Nonwritable Memory

Memory can be marked as nonwritable. Sections of memory can be write-protected using **vmStateSet( )** to prevent inadvertent access.

One use of this is to restrict modification of a data object to a particular routine. If a data object is global but read-only, tasks can read the object but not modify it. Any task that must modify this object must call the associated routine. Inside the routine, the data is made writable for the duration of the routine, and on exit, the memory is set to **VM_STATE_WRITABLE_NOT**.

Example 12-2 Nonwritable Memory

In this code example, to modify the data structure pointed to by **pData**, a task must call **dataModify( )**. This routine makes the memory writable, modifies the data, and sets the memory back to nonwritable. If a task tries to read the memory, it is successful; however, if it tries to modify the data outside of **dataModify( )**, a bus error occurs.

```
/* privateCode.h - header file to make data writable from routine only */

#define MAX 1024

typedef struct myData
    {
    char stuff[MAX];
    int moreStuff;
    } MY_DATA;

/* privateCode.c - uses VM contexts to make data private to a code segment */

#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
#include "privateCode.h"

MY_DATA * pData;
SEM_ID dataSemId;
int pageSize;

/*
 * initData - allocate memory and make it nonwritable
 *
 * This routine initializes data and should be called only once.
 *
 */

STATUS initData (void)
    {
    pageSize = vmPageSizeGet();

    /* create semaphore to protect data */

    dataSemId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);

    /* allocate memory = to a page */

    pData = (MY_DATA *) valloc (pageSize);

    /* initialize data and make it read-only */
    bzero (pData, pageSize);
    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE_NOT) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }

    /* release semaphore */

    semGive (dataSemId);
    return (OK);
    }

/*
 * dataModify - modify data
 *
 * To modify data, tasks must call this routine, passing a pointer to
 * the new data.
 * To test from the shell use:
 *     -> initData
 *     -> sp dataModify
 *     -> d pData
 *     -> bfill (pdata, 1024, 'X')
 */

STATUS dataModify
    (
    MY_DATA * pNewData
    )
    {

    /* take semaphore for exclusive access to data */

    semTake (dataSemId, WAIT_FOREVER);

    /* make memory writable */

    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }

    /* update data*/

    bcopy (pNewData, pData, sizeof(MY_DATA));

    /* make memory not writable */

    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE_NOT) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }

    semGive (dataSemId);

    return (OK);
    }
```

12.5.5 Troubleshooting

If **INCLUDE_MMU_FULL_SHOW** is included in the project facility VxWorks view, you can use **vmContextShow( )** to display a virtual memory context on the standard output device. In the following example, the current virtual memory context is displayed. Virtual addresses between 0x0 and 0x59fff are write-protected; 0xff800000 through 0xffbfffff are noncacheable; and 0x2000000 through 0x2005fff are private. All valid entries are listed and marked with a *V+*. Invalid entries are not listed.

```
-> vmContextShow 0
value = 0 = 0x0
```

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

```
VIRTUAL ADDR   BLOCK LENGTH   PHYSICAL ADDR   STATE
0x0            0x5a000        0x0             W- C+ V+ (global)
0x5a000        0x1f3c000      0x5a000         W+ C+ V+ (global)
0x1f9c000      0x2000         0x1f9c000       W+ C+ V+ (global)
0x1f9e000      0x2000         0x1f9e000       W- C+ V+ (global)
0x1fa0000      0x2000         0x1fa0000       W+ C+ V+ (global)
0x1fa2000      0x2000         0x1fa2000       W- C+ V+ (global)
0x1fa4000      0x6000         0x1fa4000       W+ C+ V+ (global)
0x1faa000      0x2000         0x1faa000       W- C+ V+ (global)
0x1fac000      0xa000         0x1fac000       W+ C+ V+ (global)
0x1fb6000      0x2000         0x1fb6000       W- C+ V+ (global)
0x1fb8000      0x36000        0x1fb8000       W+ C+ V+ (global)
0x1fee000      0x2000         0x1fee000       W- C+ V+ (global)
0x1ff0000      0x2000         0x1ff0000       W+ C+ V+ (global)
0x1ff2000      0x2000         0x1ff2000       W- C+ V+ (global)
0x1ff4000      0x2000         0x1ff4000       W+ C+ V+ (global)
0x1ff6000      0x2000         0x1ff6000       W- C+ V+ (global)
0x1ff8000      0x2000         0x1ff8000       W+ C+ V+ (global)
0x1ffa000      0x2000         0x1ffa000       W- C+ V+ (global)
0x1ffc000      0x4000         0x1ffc000       W+ C+ V+ (global)
0x2000000      0x6000         0x1f96000       W+ C+ V+
0xff800000     0x400000       0xff800000      W- C- V+ (global)
0xffe00000     0x20000        0xffe00000      W+ C+ V+ (global)
0xfff00000     0xf0000        0xfff00000      W+ C- V+ (global)
```

12.5.6 Precautions

Memory that is marked as global cannot be remapped using **vmMap( )**. To add to global virtual memory, use **vmGlobalMap( )**. For further information on adding global virtual memory, see *12.5.2 Private Virtual Memory*, p.471.

Performances of MMUs vary across architectures; in fact, some architectures may cause the system to become non-deterministic. For additional information, see the architecture-specific documentation for your hardware.

Index

IX

IX