## Contents

1 Overview .......................................................................................................................... 1
  1.1 Introduction .................................................................................................................. 1
  1.2 Related Documentation Resources ........................................................................... 2
  1.3 VxWorks Configuration and Build ........................................................................... 3

2 Applications and Processes ............................................................................................. 5
  2.1 Introduction .................................................................................................................. 6
  2.2 Configuring VxWorks For Real-time Processes ....................................................... 8
  2.3 Real-time Processes ................................................................................................... 10
    2.3.1 Real-time Process Life-Cycle ........................................................................... 12
    2.3.2 Processes and Memory .................................................................................... 14
    2.3.3 Processes and Tasks ....................................................................................... 15
    2.3.4 Processes, Inheritance, and Resource Reclamation ........................................ 16
    2.3.5 Processes and Environment Variables ........................................................... 17
    2.3.6 Processes and POSIX ...................................................................................... 18
  2.4 POSIX PSE52 Support ............................................................................................... 18
2.5 Developing VxWorks Applications ......................................................... 18
  2.5.1 Application Structure ................................................................. 19
  2.5.2 VxWorks Header Files ............................................................... 20
  2.5.3 Applications, Processes, and Tasks ........................................... 22
  2.5.4 Applications and VxWorks Kernel Component Requirements .... 24
  2.5.5 Building Applications ................................................................. 24
  2.5.6 C++ Applications .................................................................... 25
  2.5.7 Processes and Hook Routines ..................................................... 25
  2.5.8 Application APIs, System Calls, and Library Routines .............. 25
    System Calls .................................................................................. 26
    VxWorks Libraries ......................................................................... 26
    API Documentation ....................................................................... 27
  2.5.9 POSIX ....................................................................................... 27

2.6 Developing Application Libraries ....................................................... 27
  2.6.1 Library Initialization ................................................................... 28
    C++ Library Initialization .............................................................. 29
    Handling Initialization Failures .................................................... 30
  2.6.2 Library Termination ................................................................. 30
    Using atexit() for Termination Routines ....................................... 31
  2.6.3 Developing Static Libraries ...................................................... 31
  2.6.4 Developing Shared Libraries .................................................... 32
    Configuring VxWorks for Shared Libraries .................................. 36
    Building Shared Libraries and Dynamic Applications ............... 37
    VxWorks Run-time C Library libc.so .......................................... 51
    Using Plug-Ins ............................................................................. 52
    Using readelf to Examine Dynamic ELF Files .............................. 54
    Getting Runtime Information About Shared Libraries ............... 55
    Debugging Shared Libraries ......................................................... 58
    Working With Shared Libraries From a Windows Host ............... 58

2.7 Creating and Managing Shared Data Regions ..................................... 60
  2.7.1 Configuring VxWorks for Shared Data Regions ......................... 61
Round-Robin Scheduling .............................................................. 85

3.2.4 Task Creation and Management .............................................. 88
  Task Creation and Activation ...................................................... 88
  Task Stack .................................................................................. 89
  Task Stack Protection ................................................................. 90
  Task Names and IDs ................................................................. 91
  Task Creation Options .............................................................. 92
  Task Information ..................................................................... 93
  Task Deletion and Deletion Safety ............................................ 94
  Task Execution Control ............................................................ 96

3.2.5 Tasking Extensions ............................................................... 97

3.2.6 Task Error Status: errno .......................................................... 98
  A Separate errno Value for Each Task ...................................... 99
  Error Return Convention ....................................................... 99
  Assignment of Error Status Values ........................................... 99

3.2.7 Task Exception Handling ....................................................... 99

3.2.8 Shared Code and Reentrancy .................................................. 100
  Dynamic Stack Variables ......................................................... 101
  Guarded Global and Static Variables ...................................... 102
  Task Variables ......................................................................... 102
  Multiple Tasks with the Same Main Routine ......................... 103

3.3 Intertask and Interprocess Communications .............................. 105

3.3.1 Public and Private Objects .................................................... 105
  Creating and Naming Public and Private Objects .................. 106
  Object Ownership and Resource Reclamation ....................... 106

3.3.2 Shared Data Structures ....................................................... 107

3.3.3 Mutual Exclusion ............................................................... 108
  Preemptive Locks and Latency ............................................... 108

3.3.4 Semaphores ........................................................................ 109
  Semaphore Control ............................................................... 110
  Binary Semaphores ............................................................... 111
  Mutual-Exclusion Semaphores .............................................. 113
  Counting Semaphores ............................................................ 117
  Special Semaphore Options .................................................... 118
  Semaphores and VxWorks Events ......................................... 120
3.3.5 Message Queues ....................................................................................... 120
  VxWorks Message Queues ........................................................................ 121
  Displaying Message Queue Attributes ................................................... 124
  Servers and Clients with Message Queues ............................................. 124
  Message Queues and VxWorks Events ................................................. 125

3.3.6 Pipes ........................................................................................................... 125

3.3.7 VxWorks Events ....................................................................................... 126
  Preparing a Task to Receive Events ....................................................... 127
  Sending Events to a Task ........................................................................ 128
  Accessing Event Flags ............................................................................. 130
  Events Routines ........................................................................................ 131
  Task Events Register ................................................................................ 131
  Show Routines and Events ..................................................................... 132

3.3.8 Message Channels .................................................................................... 133
  Single-Node Communication with COMP ........................................... 134
  Multi-Node Communication with TIPC ............................................... 137
  Socket Name Service ............................................................................. 138
  Socket Application Libraries ................................................................. 142
  Configuring VxWorks for Message Channels ...................................... 145
  Comparing Message Channels and Message Queues ........................ 148

3.3.9 Network Communication ...................................................................... 149

3.3.10 Signals ....................................................................................................... 149
  Configuring VxWorks for Signals ........................................................... 151
  Basic Signal Routines ............................................................................. 152
  Queued Signal Routines ....................................................................... 153
  Signal Events .......................................................................................... 159
  Signal Handlers ........................................................................................ 159

3.4 Timers ........................................................................................................... 162

4 POSIX Facilities ............................................................................................ 163

4.1 Introduction .................................................................................................. 164

4.2 Configuring VxWorks with POSIX Facilities .......................................... 166
  4.2.1 POSIX PSE52 Support ...................................................................... 166
  4.2.2 VxWorks Components for POSIX Facilities .................................... 168
4.3  General POSIX Support ........................................................................................................ 169
4.4  Standard C Library: Libc .................................................................................................... 171
4.5  POSIX Header Files ............................................................................................................ 172
4.6  POSIX Namespace ................................................................................................................ 174
4.7  POSIX Process Support ...................................................................................................... 176
4.8  POSIX Clocks and Timers ................................................................................................... 177
4.9  POSIX Asynchronous I/O ................................................................................................... 181
4.10 POSIX Advisory File Locking ............................................................................................ 181
4.11 POSIX Page-Locking Interface .......................................................................................... 182
4.12 POSIX Threads .................................................................................................................. 183
  4.12.1 POSIX Thread Stack Guard Zones ................................................................. 183
  4.12.2 POSIX Thread Attributes ...................................................................................... 184
  4.12.3 VxWorks-Specific Pthread Attributes ............................................................ 184
  4.12.4 Specifying Attributes when Creating Pthreads ........................................... 185
  4.12.5 POSIX Thread Creation and Management ................................................... 187
  4.12.6 POSIX Thread Attribute Access .......................................................................... 188
  4.12.7 POSIX Thread Private Data ................................................................................ 189
  4.12.8 POSIX Thread Cancellation ................................................................................ 190
4.13 POSIX Thread Mutexes and Condition Variables .......................................................... 192
  4.13.1 Thread Mutexes ....................................................................................................... 192
     Type Mutex Attribute ................................................................................................. 193
     Protocol Mutex Attribute .......................................................................................... 194
     Priority Ceiling Mutex Attribute ............................................................................. 195
  4.13.2 Condition Variables ............................................................................................... 195
4.14 POSIX and VxWorks Scheduling ....................................................................................... 197
  4.14.1 Differences in POSIX and VxWorks Scheduling ........................................... 199
4.14.2 POSIX and VxWorks Priority Numbering ............................................ 199
4.14.3 Default Scheduling Policy ................................................................. 200
4.14.4 VxWorks Traditional Scheduler ......................................................... 201
4.14.5 POSIX Threads Scheduler ................................................................. 202
4.14.6 POSIX Scheduling Routines ............................................................... 206
4.14.7 Getting Scheduling Parameters: Priority Limits and Time Slice ...... 207

4.15 POSIX Semaphores .................................................................................. 208
  4.15.1 Comparison of POSIX and VxWorks Semaphores ......................... 209
  4.15.2 Using Unnamed Semaphores ............................................................ 210
  4.15.3 Using Named Semaphores .............................................................. 213

4.16 POSIX Message Queues .......................................................................... 218
  4.16.1 Comparison of POSIX and VxWorks Message Queues .................... 220
  4.16.2 POSIX Message Queue Attributes .................................................. 220
  4.16.3 Communicating Through a Message Queue ..................................... 222
  4.16.4 Notification of Message Arrival ....................................................... 226

4.17 POSIX Signals ......................................................................................... 232

4.18 POSIX Memory Management ................................................................. 232
  4.18.1 POSIX Memory Management APIs .................................................. 232
  4.18.2 Anonymous Memory Mapping ....................................................... 234
  4.18.3 Shared Memory Objects .................................................................. 236
  4.18.4 Memory Mapped Files .................................................................... 237
  4.18.5 Memory Protection ......................................................................... 238
  4.18.6 Memory Locking ............................................................................ 238

4.19 POSIX Trace .......................................................................................... 239
  Trace Operation .......................................................................................... 240
  Trace APIs ................................................................................................. 240
  Trace Code and Record Example .............................................................. 242
5 Memory Management ................................................................. 245
  5.1 Introduction .............................................................................. 245
  5.2 VxWorks Component Requirements ........................................ 246
  5.3 Heap and Memory Partition Management ................................. 246
  5.4 Memory Error Detection .......................................................... 248
    5.4.1 Heap and Partition Memory Instrumentation ...................... 248
    5.4.2 Compiler Instrumentation ................................................. 255

6 I/O System ................................................................................... 259
  6.1 Introduction .............................................................................. 260
  6.2 Files, Devices, and Drivers ...................................................... 261
    6.2.1 Filenames and the Default Device ...................................... 262
  6.3 Basic I/O .................................................................................. 263
    6.3.1 File Descriptors ............................................................... 264
    6.3.2 Standard Input, Standard Output, and Standard Error .......... 265
    6.3.3 Standard I/O Redirection .................................................. 265
    6.3.4 Open and Close ............................................................... 267
    6.3.5 Create and Remove ......................................................... 269
    6.3.6 Read and Write ............................................................... 270
    6.3.7 File Truncation ............................................................... 271
    6.3.8 I/O Control ..................................................................... 271
    6.3.9 Pending on Multiple File Descriptors: The Select Facility ...... 272
    6.3.10 POSIX File System Routines .............................................. 274
  6.4 Buffered I/O: stdio ................................................................. 274
    6.4.1 Using stdio ........................................................................ 275
6.4.2 Standard Input, Standard Output, and Standard Error ..................... 276
6.5 Other Formatted I/O .............................................................................................. 276
6.6 Asynchronous Input/Output ................................................................................ 277
   6.6.1 The POSIX AIO Routines ........................................................................ 277
   6.6.2 AIO Control Block .................................................................................... 278
   6.6.3 Using AIO .................................................................................................. 279
       Alternatives for Testing AIO Completion ..................................................... 280
6.7 Devices in VxWorks ............................................................................................... 280
   6.7.1 Serial I/O Devices: Terminal and Pseudo-Terminal Devices .......... 280
       tty Options ................................................................................................. 281
       Raw Mode and Line Mode ........................................................................ 282
       tty Special Characters .............................................................................. 282
   6.7.2 Pipe Devices ............................................................................................. 283
       Creating Pipes ........................................................................................... 284
       I/O Control Functions ............................................................................. 284
   6.7.3 Pseudo I/O Device .................................................................................. 284
       I/O Control Functions ............................................................................. 285
   6.7.4 Network File System (NFS) Devices ..................................................... 285
       I/O Control Functions for NFS Clients .................................................... 286
   6.7.5 Non-NFS Network Devices .................................................................... 286
       I/O Control Functions ............................................................................. 287
   6.7.6 Sockets ....................................................................................................... 287
   6.7.7 Transaction-Based Reliable File System Facility: TRFS ............. 287
       Configuring VxWorks With TRFS .......................................................... 288
       Automatic Instantiation of TRFS .......................................................... 288
       Using TRFS in Applications ................................................................. 289
       TRFS Code Example ............................................................................... 289
# Local File Systems

## 7.1 Introduction

## 7.2 File System Monitor

## 7.3 Virtual Root File System: VRFS

## 7.4 Highly Reliable File System: HRFS

### 7.4.1 Configuring VxWorks for HRFS

### 7.4.2 HRFS and POSIX PSE52

### 7.4.3 Creating an HRFS File System

### 7.4.4 Transactional Operations

### 7.4.5 File Access Time Stamps

### 7.4.6 Maximum Number of Files and Directories

### 7.4.7 Working with Directories

#### Creating Subdirectories

#### Removing Subdirectories

#### Reading Directory Entries

### 7.4.8 Working with Files

#### File I/O Routines

#### File Linking and Unlinking

#### File Permissions

### 7.4.9 Crash Recovery and Volume Consistency

### 7.4.10 I/O Control Functions Supported by HRFS

## 7.5 MS-DOS-Compatible File System: dosFs

### 7.5.1 Configuring VxWorks for dosFs

### 7.5.2 Creating a dosFs File System

### 7.5.3 Working with Volumes and Disks

#### Accessing Volume Configuration Information

#### Synchronizing Volumes

### 7.5.4 Working with Directories

#### Creating Subdirectories
Removing Subdirectories ................................................................. 307
Reading Directory Entries ............................................................. 308
7.5.5 Working with Files ................................................................. 308
File I/O Routines ........................................................................... 308
File Attributes ................................................................................ 308
7.5.6 Disk Space Allocation Options ................................................. 310
Choosing an Allocation Method ...................................................... 311
Using Cluster Group Allocation ...................................................... 311
Using Absolutely Contiguous Allocation ...................................... 312
7.5.7 Crash Recovery and Volume Consistency ............................... 314
7.5.8 I/O Control Functions Supported by dosFsLib ....................... 314
7.5.9 Booting from a Local dosFs File System Using SCSI ............... 316
7.6 Raw File System: rawFs ............................................................. 316
7.6.1 Configuring VxWorks for rawFs ............................................. 317
7.6.2 Creating a rawFs File System .................................................. 317
7.6.3 Mounting rawFs Volumes ...................................................... 317
7.6.4 rawFs File I/O ...................................................................... 318
7.6.5 I/O Control Functions Supported by rawFsLib ....................... 318
7.7 CD-ROM File System: cdromFs ................................................ 319
7.7.1 Configuring VxWorks for cdromFs ........................................ 321
7.7.2 Creating and Using cdromFs .................................................. 321
7.7.3 I/O Control Functions Supported by cdromFsLib ................. 322
7.7.4 Version Numbers ................................................................. 323
7.8 Read-Only Memory File System: ROMFS ................................. 323
7.8.1 Configuring VxWorks with ROMFS ..................................... 324
7.8.2 Building a System With ROMFS and Files ............................ 324
7.8.3 Accessing Files in ROMFS ................................................... 324
7.8.4 Using ROMFS to Start Applications Automatically ............... 325
7.9 Target Server File System: TSFS ................................................................. 325
    Socket Support ......................................................................................... 326
    Error Handling ......................................................................................... 327
    Configuring VxWorks for TSFS Use ...................................................... 327
    Security Considerations ....................................................................... 327
    Using the TSFS to Boot a Target .......................................................... 328

8 Error Detection and Reporting ............................................................ 329

8.1 Introduction .......................................................................................... 329

8.2 Configuring Error Detection and Reporting Facilities ....................... 330
    8.2.1 Configuring VxWorks ..................................................................... 330
    8.2.2 Configuring the Persistent Memory Region .................................... 331
    8.2.3 Configuring Responses to Fatal Errors ........................................... 332

8.3 Error Records ....................................................................................... 332

8.4 Displaying and Clearing Error Records .............................................. 334

8.5 Fatal Error Handling Options ............................................................... 335
    8.5.1 Configuring VxWorks with Error Handling Options ..................... 336
    8.5.2 Setting the System Debug Flag ....................................................... 336
    Setting the Debug Flag Staticaly ............................................................ 337
    Setting the Debug Flag Interactively ..................................................... 337

8.6 Other Error Handling Options for Processes ..................................... 338

8.7 Using Error Reporting APIs in Application Code .............................. 339

8.8 Sample Error Record ........................................................................... 339

9 C++ Development ................................................................................. 341

9.1 Introduction ......................................................................................... 341

9.2 C++ Code Requirements .................................................................... 342
9.3 C++ Compiler Differences ................................................................. 342
  9.3.1 Template Instantiation ............................................................... 343
  9.3.2 Exception Handling ................................................................. 344
  9.3.3 Run-Time Type Information ...................................................... 345

9.4 Namespaces ...................................................................................... 345

9.5 C++ Demo Example .......................................................................... 346

Index ..................................................................................................... 347
1.1 Introduction

This manual describes the VxWorks operating system, and how to use VxWorks facilities in the development of real-time applications and systems. It covers the following topics:

- real-time process (RTP) applications and process management
- multitasking facilities
- POSIX standard interfaces
- memory management
- I/O system
- local file systems
- error detection and reporting
- C++ development

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

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

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

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

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

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

1.3 VxWorks Configuration and Build

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

**NOTE:** In this book, as well as in the VxWorks API references, VxWorks components are identified by the names used in component description files, which is in the form of `INCLUDE_FOO`. Similarly, configuration parameters are identified by their configuration parameter names, such as `NUM_FOO_FILES`.

Component and parameter names can be used directly to identify components and configure VxWorks if you work with the command-line configuration facilities. Wind River Workbench provides fuller descriptions of the components and their parameters in the GUI. But you can also use a simple search facility to locate a component based on its component name. Once you have located the component, you can access the component’s parameters through the GUI.
2

Applications and Processes

2.1 Introduction 6
2.2 Configuring VxWorks For Real-time Processes 8
2.3 Real-time Processes 10
2.4 POSIX PSE52 Support 18
2.5 Developing VxWorks Applications 18
2.6 Developing Application Libraries 27
2.7 Creating and Managing Shared Data Regions 60
2.8 Executing Applications 63
2.9 Bundling Applications with a VxWorks System using ROMFS 74
2.1 Introduction

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

- Occupy continuous blocks of virtual memory.
- Are started with a spawn model, in which the instantiation of the process is separate from the loading of the application.
- Load applications in their entirety; there is no demand paging.
- Maintain the VxWorks task scheduling model, in which all tasks are scheduled globally (processes themselves are not scheduled).
- Provide traditional VxWorks APIs in addition to POSIX APIs.

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

VxWorks real-time processes provide the means for executing applications in user mode. Each process has its own address space, which contains the executable program, the program’s data, stacks for each task, the heap, and resources associated with the management of the process itself (such as memory-allocation tracking).

Many processes may be present in memory at once, and each process may contain more than one task (sometimes known as a thread in other operating systems). Each VxWorks process has its own region of virtual memory; processes do not overlap in virtual memory. This flat virtual-memory map provides advantages in speed, in a programming model that accommodates systems with and without an MMU, and in debugging applications.

There is no limit to the number of processes in a system, or to the number of tasks in a process, other than the limit imposed by the amount of available memory.

¹. VxWorks can be configured to provide POSIX PSE52 support for individual processes.
Each real-time process (RTP) application can be created as a fully linked or partially linked (for use with shared libraries), relocatable executable with cross-development tools on a host system. Applications are built as single executable files independent of the operating system, with the user code being linked to the required VxWorks application API libraries.

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

The VxWorks operating system is configured and built independently of any applications that it might execute. Applications are built separately, and they can either be stored separately or bundled with VxWorks in an additional build step that combines the operating system and applications into a single system image (using the ROMFS file system). VxWorks need only be configured with the appropriate components for real-time process support and any other facilities required by the application (for example, message queues). This independence of operating system from applications allows for development of a variety of systems, using differing applications, that are based on a single operating system configuration. That is, a single variant of the operating system can be combined with different sets of applications to create different systems. The operating system does not need to be aware of what applications it will run before it is configured and built, as long as its configuration includes the components required to support the applications in question.

The isolation of applications in processes effectively prevents symbol name clashes during integration. As processes are completely linked, they never import functions from outside the process even when external functions have the same name. The name and symbol spaces of the kernel and processes are isolated.

Executable application files can be stored on disks, in RAM, flash, or ROM. They can be stored on the target or anywhere else that is accessible over a network connection. The executables are loaded from a file system; and any file system for which the kernel has support (ROMFS, NFS, ftp, and so on) can be used. The ROMFS file system technology is particularly useful for deployed systems. It allows developers to bundle application executables with the VxWorks image into a single file that the boot loader can load from ROM. Unlike other operating systems, no root file system—such as on NFS or a diskette—is required to hold application binaries, configuration files, and so on.
In order to run RTP applications on a hardware target, VxWorks must be configured with the \texttt{INCLUDE\_RTP} component and rebuilt.

Note that if a system is configured with \texttt{INCLUDE\_RTP} the MMU components required for memory protection are included by default. To create a system with processes, but without MMU support, the MMU components must be removed from the VxWorks configuration.

The default image provided for the VxWorks simulator includes all the necessary components.
Additional Components Options

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

- **INCLUDE_ROMFS** for the ROMFS file system.
- **INCLUDE_RTP_APPL_USER, INCLUDE_RTP_APPL_INIT_STRING, INCLUDE_RTP_APPL_INIT_BOOTLINE, and INCLUDE_RTP_APPL_INIT_CMD_SHELL_SCRIPT** for various ways of automatically starting applications at boot time.
- **INCLUDE_SHARED_DATA** for shared data regions.
- **INCLUDE_SHL** for shared libraries.
- **INCLUDE_RTP_HOOKS** for the programmatic hook facility, which allows for registering kernel routines that are to be executed at various points in a process’s life-cycle.
- **INCLUDE_POSIX_PTHREAD_SCHEDULER and INCLUDE_POSIX_CLOCK** for POSIX thread support. This replaces the traditional VxWorks scheduler with a scheduler handling user threads in a manner consistent with POSIX.1. VxWorks tasks and kernel pthreads are handled as usual. Note that the **INCLUDE_POSIX_PTHREAD_SCHEDULER** is required for using pthreads in processes. For more information, see 4.14 POSIX and VxWorks Scheduling, p.197.
- **INCLUDE_PROTECT_TASK_STACK** for stack protection. For deployed systems this component may be omitted to save on memory usage. See Task Stack, p.89 for more information.

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

- The various **INCLUDE_SHELL_feature** components for the kernel shell, which, although not required for applications and processes, are needed for running applications from the command line, executing shell scripts, and on-target debugging.
- The **INCLUDE_WDB** component for using the host tools.
- Either the **INCLUDE_NET_SYM_TBL** or the **INCLUDE_STANDALONE_SYM_TBL** component, which specify whether symbols for the shell are loaded or built-in.
- The **INCLUDE_DISK_UTIL** and **INCLUDE_RTP_SHOW** components, which include useful shell routines.
For information about the kernel shell, symbol tables, and show routines, see the
VxWorks Kernel Programmer’s Guide: Target Tools. For information about the host
shell, see the VxWorks Command-Line Tools User’s Guide.

Component Bundles

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

- **BUNDLE_RTP_DEPLOY** is designed for deployed systems (final products), and
  is composed of INCLUDE_RTP, INCLUDE_RTP_APPL, INCLUDE_RTP_HOOKS,
  INCLUDE_SHARED_DATA, and the BUNDLE_SHL components.

- **BUNDLE_RTP_DEVELOP** is designed for the development environment, and is
  composed of BUNDLE_RTP_DEPLOY, INCLUDE_RTP_SHELL_CMD,
  INCLUDE_RTP_SHOW, INCLUDE_SHARED_DATA_SHOW,
  INCLUDE_SHL_SHOW, INCLUDE_RTP_SHOW_SHELL_CMD,
  INCLUDE_SHL_SHELL_CMD, components.

- **BUNDLE_RTP_POSIX_PSE52** provides POSIX PSE52 support for individual
  processes (for more information see 4.2 Configuring VxWorks with POSIX
  Facilities, p.166). It can be used with either BUNDLE_RTP_DEPLOY or
  BUNDLE_RTP_DEVELOP.

Configuration and Build

For information about configuring and building VxWorks, see the Wind River

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

### 2.3 Real-time Processes

A common definition of a process is “a program in execution,” and VxWorks
processes are no different in this respect. VxWorks processes, however, are called
real-time processes precisely because they are designed to support the
determinism required of real-time systems. They do so in various ways:
2 Applications and Processes
2.3 Real-time Processes

- Processes are not scheduled—tasks are scheduled globally throughout the system.
- Processes can be preempted in kernel mode as well as in user mode. Every task has both a user mode and a kernel mode stack. (The VxWorks kernel is fully preemptive.)
- Process creation takes place in two phases—each with a separate task priority—which separates instantiation from loading the application. The second task, therefore, bears the cost of instantiation itself, at its own task priority level. The calling task is not impacted, unless it is coded to wait.
- Processes load applications in their entirety; there is no demand paging.

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

By way of contrast, the scheduling policy for non-real-time systems is based on time-sharing, as well as a dynamic determination of process priority that ensures that no process is denied use of the CPU for too long, and that no process monopolizes the CPU. Each Linux process is allocated its own time quantum, which it expends on its own tasks, regardless of the priorities of tasks in other processes. Thus, for example, a high priority task in process A may pend for a significant amount of time while processes B and C use up their time quantum on their own (lower priority) tasks. By way of analogy, it is something like spending small and equal amounts of time on each plate in a gourmet prix fixe meal in rotation—without regard to whether the hot ones get cold or the cold ones warm, and without regard to the gastronomic intentions of the chef’s artistry.

VxWorks does provide an optional time-sharing capability—round-robin scheduling—but it does not interfere with priority-based preemption, and is therefore deterministic. Round-robin scheduling simply ensures that when there is more than one task with the highest priority ready to run at the same time, the CPU is shared between those tasks. No one of them, therefore, can usurp the processor until it is blocked. (For more information about VxWorks scheduling see 3.2.2 Task Scheduling, p.83.)

The manner in which VxWorks processes are created also supports the determinism required of real-time systems. The creation of a VxWorks process takes place in two distinct phases, and the executable is loaded in its entirety when the process is created. In the first phase, the rtpSpawn() call creates the process
object in the system, allocates virtual and physical memory to it, and creates the initial process task (see 2.3.3 Processes and Tasks, p.15). In the second phase, the initial process task loads the entire executable and starts the main routine.

This approach provides for system determinism in two ways:

- First, the work of process creation is divided between the `rtpSpawn()` task and the initial process task—each of which has its own distinct task priority. This means that the activity of loading applications does not occur at the priority, or with the CPU time, of the task requesting the creation of the new process. Therefore, the initial phase of starting a process is discrete and deterministic, regardless of the application that is going to run in it. And for the second phase, the developer can assign the task priority appropriate to the significance of the application, or to take into account necessarily indeterministic constraints on loading the application (for example, if the application is loaded from networked host system, or local disk). The application is loaded with the same task priority as the priority with which it will run. In a way, this model is analogous to asynchronous I/O, as the task that calls `rtpSpawn()` just initiates starting the process and can concurrently perform other activities while the application is being loaded and started.

- Second, the entire application executable is loaded when the process is created, which means that the determinacy of its execution is not compromised by incremental loading during execution. This feature is obviously useful when systems are configured to start applications automatically at boot time—all executables are fully loaded and ready to execute when the system comes up.

Note that `rtpSpawn()` has an option that provides for synchronizing for the successful loading and instantiation of the new process.

Note that the creation of VxWorks processes involves no copying or sharing of the parent processes page frames (copy-on-write), as is the case with some versions of UNIX and Linux. The flat virtual-memory model provided by VxWorks prohibits this approach. For information about the issue of inheritance of attributes from parent processes, see 2.3.4 Processes, Inheritance, and Resource Reclamation, p.16.

### 2.3.1 Real-time Process Life-Cycle

The life-cycle of VxWorks real-time processes is largely consistent with the POSIX process model.

VxWorks processes can be started in a variety of ways:

- interactively from the kernel shell
Applications and Processes
2.3 Real-time Processes

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

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

Each process can execute one or more tasks. When a process is created, the system spawns a single task to initiate execution of the application. The application may then spawn additional tasks to perform various functions.

The creation of a process includes several phases, as described in 2.3 Real-time Processes, p.10. For information about what operations are possible on a process in each phase of its instantiation, see the API reference for rtpLib.

Processes are terminated under the following circumstances:

- When the last task in the process exits.
- If any task in the process calls exit(), regardless of whether or not other tasks are running in the process.
- If the process’ main() routine returns.
  This is because exit() is called implicitly when main() returns. An application in which main() spawns tasks can be written to avoid this behavior—and to allow its other tasks to continue operation—by including a taskExit() call as the last statement in main(). See 2.5 Developing VxWorks Applications, p.18.
- If the kill() routine is used to terminate the process.
- If rtpDelete() is called on the process—from a program, a kernel module, the C interpreter of the shell, or from Workbench. Or if the rtp delete command is used from the shell’s command interpreter.
- If a process takes an exception during its execution.
  This default behavior can be changed for debugging purposes. When the error detection and reporting facilities are included in the system, and they are set to debug mode, processes are not terminated when an exception occurs.

Note that if a process fails while the shell is running, a message is printed to the shell console. Error messages can be recorded with the VxWorks error detection and reporting facilities (see 8. Error Detection and Reporting).
For information about attribute inheritance and what happens to a process’ resources when it terminates, see 2.3.4 Processes, Inheritance, and Resource Reclamation, p.16.

2.3.2 Processes and Memory

Each process has its own address space, which contains the executable program, the program’s data, stacks for each task, the heap, and resources associated with the management of the process itself (such as local heap management).

Many processes may be present in memory at once. Each VxWorks process has its own region of virtual memory; processes do not overlap in virtual memory. This flat virtual-memory map provides the following advantages:

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

Systems can be developed and debugged on targets with an MMU, and then shipped on hardware that does not have one, or has one that is not enabled for deployment. The advantages of being able to do so include facilitating debugging in development, lower cost of shipped units, as well as footprint and performance advantages of targets without an MMU (or with one that is not enabled).

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

As processes run a fully linked image without external references, a process cannot call a routine in another process, or a kernel routine that is not exported as a function call—whether or not the MMU is enabled. However, if the MMU is not enabled, a process can read and write memory external to its own address space, and could cause the system to malfunction.
While the address space of each process is invisible to tasks running in other processes, tasks can communicate across process boundaries through the use of various IPC mechanisms (including public semaphores, public message queues, and message channels) and shared data memory regions. See 3.3 Intertask and Interprocess Communications, p.105 and 2.7 Creating and Managing Shared Data Regions, p.60 for more information.

At startup time, the resources internally required for the process (such as the heap) are allocated on demand. The application’s text is guaranteed to be write-protected, and the application’s data readable and writable, as long as an MMU is present and the operating system is configured to manage it. While memory protection is provided by MMU-enforced partitions between processes, there is no mechanism to provide resource protection by limiting memory usage of processes to a specified amount. For more information, see 5. Memory Management.

2.3.3 Processes and Tasks

Each VxWorks process may contain more than one task. When a process is created, an initial task is spawned to begin execution of the application. The name of the process’s initial task is based on the name of the executable file, with the following modifications:

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

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

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

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

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

Providing heap management in user space provides for speed and improved performance because the application does not incur the overhead of a system call for memory during its execution. However, if the heap is exhausted the system
automatically allocates more memory for the process (by default), in which case a system call is made. Environment variables control whether or not the heap grows (see 5.3 Heap and Memory Partition Management, p.246).

For more information, see 3. Multitasking.

2.3.4 Processes, Inheritance, and Resource Reclamation

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

VxWorks processes inherit certain attributes of their parent. The child process inherits the file descriptors of its parent, which means that they can access the same files (if they are open), signal masks, and environment variables.

By default, when a process is terminated, and its parent is not the kernel, it becomes a zombie process.2 The parent process can then use wait() or waitpid() to get the exit status of the child process. After it has done so, the zombie entity is deleted automatically. The default behavior of zombie creation can be prevented by either of these two methods:

- By ignoring the SIGCHLD signal. That is, by setting the SIGCHLD signal handler to SIG_IGN with sigaction().
- By hooking a SIGCHLD handling with SA_NOCLDSTOP | SA_NOCLDWAIT sa_flag set with sigaction().

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

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

---

2. A zombie process is a “process that has terminated and that is deleted when its exit status has been reported to another process which is waiting for that process to terminate.” (The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition.)
When a process terminates, all resources owned by the process (objects, data, and so on) are returned to the system. The resources used internally for managing the process are released, as are all resources owned by the process. All information about that process is eliminated from the system (with the exception of any temporary zombie process information). Resource reclamation ensures that all resources that are not in use are immediately returned to the system and available for other uses.

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

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

For information about object ownership, and about public and private objects, see 3.3.1 Public and Private Objects, p.105.

### 2.3.5 Processes and Environment Variables

By default, a process is created without environment variables. However, the creator of a process can pass an environment array to the application (for example, when a process is started from the shell’s command interpreter, it always passes the SHELL_INTERPRETER variable). See the rtpSpawn() API reference and 2.5.1 Application Structure, p.19 for details.

In addition, if the creator is a kernel task, the contents of that task’s environment array can be duplicated in the application’s environment array using the getenv() routine.

A task in a process (or in an application library) can create, reset, and remove environment variables in a process. The getenv() routine can be used to get the environment variables, and the setenv() and unsetenv() routines to change or remove them. They environment array can also be manipulated directly.
In a manner consistent with the POSIX standard, all tasks in a process share the same environment variables—unlike kernel tasks, which each have their own set of environment variables.

### 2.3.6 Processes and POSIX

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

- an overlapping virtual memory model
- process creation with `fork()` and `exec()`
- memory-mapped files
- file ownership and file permissions

For information about POSIX support, see 4. POSIX Facilities.

### 2.4 POSIX PSE52 Support

VxWorks can be configured to provide POSIX PSE52 support (for individual processes, as defined by the profile). For detailed information, see 2.2 Configuring VxWorks For Real-time Processes, p.8, 4.1 Introduction, p.164, 4.2 Configuring VxWorks with POSIX Facilities, p.166, and 7.4.2 HRFS and POSIX PSE52, p.298.

### 2.5 Developing VxWorks Applications

VxWorks RTP applications are created as a fully linked, relocatable executables. They are built independently of the VxWorks operating system, using cross-development tools on the host system. When an application is built, user code is linked to the required VxWorks application API libraries, and an ELF executable is produced. By convention, VxWorks RTP executables are named with
2 Applications and Processes
2.5 Developing VxWorks Applications

A VxWorks application can be loaded from any file system for which the kernel has support (NFS, ftp, and so on). In addition, applications can be bundled into a single image with the operating system (see 2.9 Bundling Applications with a VxWorks System using ROMFS, p.74). Executable application files can be stored on disks, in RAM, flash, or ROM.

2.5.1 Application Structure

VxWorks applications have a simple structural requirement that is common to C programs on other operating systems—they must include a main() routine. The main() routine can be used with the conventional argc and argv arguments, as well as two additional optional arguments, envp and auxp:

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

The envp and auxp arguments are usually not required by the application code.

The envp argument is used for passing VxWorks environment variables to the application. These can be set by a user and are typically inherited from the calling environment. Note that the getenv() routine can be used to get the environment variables programmatically, and the setenv() and unsetenv() routines to change or remove them. (See 2.3.5 Processes and Environment Variables, p.17.)

Environment variables are general properties of the running system, such as the default path—unlike argv arguments, which are passed to a particular invocation of the application, and are unique to that application. The system uses the auxp vector to pass system information to the new process, including page size, cache alignment size and so on.

The argv[0] argument is typically the relative path to the executable.
2.5.2 VxWorks Header Files

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

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

VxWorks system header files for RTP applications are in the directory `installDir/vxworks-6.x/target/usr/h` and its subdirectories (different directories are used for kernel applications).

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

**POSIX Header Files**

Traditionally, VxWorks has provided many header files that are described by POSIX.1, although their content was only partially that described by POSIX.1. For user-mode applications the POSIX header files are more strictly compliant with the POSIX.1 description, in both in their content and in their location. See 4.5 POSIX Header Files, p.172 for more information.

**VxWorks Header File: vxWorks.h**

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

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

**Other VxWorks Header Files**

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

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

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

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

ANSI C++ Header Files

Each compiler has its own C++ libraries and C++ headers (such as iostream and new). The C++ headers are located in the compiler installation directory rather than in installDir/vxworks-6.x/target/usr/h. No special flags are required to enable the compilers to find these headers. For more information about C++ development, see 9. C++ Development.

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

The -I Compiler Flag

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

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

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

#include <vxWorks.h>
#include <sys/stat.h>

VxWorks Nested Header Files

Some VxWorks facilities make use of other, lower-level VxWorks facilities. For example, the tty management facility uses the ring buffer subroutine library. The tty header file tyLib.h uses definitions that are supplied by the ring buffer header file rngLib.h.
It would be inconvenient to require you to be aware of such include-file interdependencies and ordering. Instead, all VxWorks header files explicitly include all prerequisite header files. Thus, *tyLib.h* itself contains an include of *rngLib.h*. (The one exception is the basic VxWorks header file *vxWorks.h*, which all other header files assume is already included.)

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

**VxWorks Private Header Files**

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

Some header files mark internal details using HIDDEN comments:

```c
/* HIDDEN */
```

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

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

**2.5.3 Applications, Processes, and Tasks**

A process is an instance of an application in execution. A process must be spawned in order to initiate execution of an application; when the application exits, the process terminates.

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

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

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

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

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

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

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

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

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

For information about task priorities and synchronization, see 3.2 Tasks and Multitasking, p.78 and 3.3 Intertask and Interprocess Communications, p.105.
2.5.4 Applications and VxWorks Kernel Component Requirements

VxWorks is a highly configurable operating system. Because RTP applications are built independently of the operating system, the build process cannot determine if the instance of VxWorks on which the application will eventually run has been configured with all of the components that the application requires (for example, networking and file systems).

It is, therefore, important for application code to check for errors indicating that kernel facilities are not available (that is, check the return values of API calls) and to respond appropriately. If an API requires a facility that is not configured into the kernel, an errno value of ENOSYS is returned when the API is called.

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

2.5.5 Building Applications

The VxWorks cross-development environment provides simple mechanisms for building applications, including a useful set of default makefile rules. Both Workbench and command line can be used to build applications.

For command-line use, the VxWorks Development Shell or wrenv utility program can be used to open a command shell with the appropriate environment variables set. See the VxWorks Command-Line Tools User’s Guide.

An application can then be compiled for VxWorks by creating a simple makefile and executing a make command in the directory that contains the application C file. Makefile macros allow for compiling the same source code for different architectures, with different compilers, and so on. For example, assuming that c:\vxApp\myVxApp.c has been created, a makefile with the following macro assignments and include statement could be used to generate the executable:

EXE = myVxApp.vxe
OBJS = myVxApp.o
include $(WIND_USR)/make/rules.rtp

When make is run, the executable is created in installDir\vxworks-6.x\target\usr\root\cpuTool\bin. As the defaults in this regard are PowerPC and the Wind River (Diab) compiler, the path and filename would be as follows:

c:\myInstallDir\vxworks-6.x\target\usr\root\PPC32diab\bin\myVxApp.vxe
Macro assignments can be used at the command line to identify other architectures and compilers. For example, the command used to build the application with the GNU compiler for Pentium 2 would be as follows:

```
make CPU=PENTIUM2 TOOL=gnu
```

Note that applications that make use of share libraries or plug-ins must be built as dynamic executables. See *Building Shared Libraries and Dynamic Applications*, p.37 for information about dynamic executables and additional `make` macros.

For information about build options, see the *VxWorks Architecture Supplement* for the target architecture in question. For information about using makefiles to build applications, see the *Wind River Command-line User’s Guide*.

### 2.5.6 C++ Applications

For information about developing C++ applications, see *9. C++ Development*.

### 2.5.7 Processes and Hook Routines

For information about hook routines, see the *VxWorks API reference* for `rtpHookLib` and *3.2.5 Tasking Extensions*, p.97.

### 2.5.8 Application APIs, System Calls, and Library Routines

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

System calls provide access to kernel facilities that are otherwise inaccessible in user space, such as APIs that involve interaction with the hardware, I/O, and the processor itself. Some library routines include system calls, and others execute entirely in user space.

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

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

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

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

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

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

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

VxWorks Libraries

VxWorks distributions include libraries of routines that provide APIs for RTP applications. Some of these routines execute entirely in the process in user mode. Others are wrapper routines that make one or more system calls, or that add additional functionality to one or more system calls. For example, printf() is a wrapper that calls the system call write(). The printf() routine performs a lot of formatting and so on, but ultimately must call write() to output the string to a file descriptor.
Library routines that do not include system calls execute in entirely user mode, and are therefore more efficient than system calls, which include the overhead of a trap to the kernel.

The standard C and C++ libraries for VxWorks applications are provided by Dinkumware, Ltd.

Note that the user-mode libraries provided for RTP applications are completely separate from kernel libraries.

For information about creating custom user-mode libraries for applications, see 2.6 Developing Application Libraries, p.27.

API Documentation

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

2.5.9 POSIX

For information about POSIX APIs available with VxWorks, and a comparison of native VxWorks and POSIX APIs, see 4. POSIX Facilities.

2.6 Developing Application Libraries

Developers can create their own custom libraries to support their RTP applications. These libraries can be developed for either static or dynamic (run-time) linking. See 2.6.4 Developing Shared Libraries, p.32 for a discussion of the differences between static and shared libraries.

As with applications, libraries should check for errors indicating that kernel facilities are not available (see 2.5.4 Applications and VxWorks Kernel Component Requirements, p.24).

The subsections 2.6.1 Library Initialization, p.28 and 2.6.2 Library Termination, p.30 provide information relevant to both static and shared libraries.
2.6.1 Library Initialization

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

If an initialization routine is required for the library, its prototype should follow this convention:

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

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

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

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

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

```c
_WRS_CONSTRUCTOR (scarceResourceInit, 101)
{
    /* Note: a FIFO mutex is preferable to a priority-based mutex
        * since task priority should not play a role in accessing the scarce
        * resource.
        */
    if ((scarceResourceMutex = semMCreate (SEM_DELETE_SAFE | SEM_Q_FIFO |
                                             SEM_USER)) == NULL)
        EDR_USR_FATAL_INJECT (FALSE,
                               "Cannot enable task protection on scarce resource\n");
}
```

(For information about using the error detection and reporting macro `EDR_USR_FATAL_INJECT`, see 8.7 Using Error Reporting APIs in Application Code, p.339.)

The rank number is used by the compiler to order the initialization routines. The rank number is referred to somewhat misleadingly as a `priority` number by the
toolchain. Rank numbers from 100 to 65,535 can be used—numbers below 100 are reserved for VxWorks libraries. Using a rank number below 100 does not have detrimental impact on the kernel, but may disturb or even prevent the initialization of the application environment (which involves creating resources such as the heap, semaphores, and so on).

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

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

C++ Library Initialization

Application libraries written in C++ may require initialization of static constructors for any global objects the library might use, in addition to the library initialization required for libraries written in C (described in 2.6.1 Library Initialization, p.28).

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

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

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

- Check whether the `ENOSYS` errno has been set, and respond appropriately. For system calls, this errno indicates that the required support component has not been included in the kernel.
- Release all the resources that have been created or obtained by the initialization routine.
- Use the `EDR_USR_FATAL_INJECT` macro to report the error. If the system has been configured with the error detection and reporting facility, the error is recorded in the error log (and the system otherwise responds to the error depending on how the facility has been configured). If the system has not been configured with the error detection and reporting facility, it attempts to print the message to a host console by way of a serial line. It is best to use the macro for this purpose without tracing. For example:

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

For more information, see 8.7 Using Error Reporting APIs in Application Code, p.339.

2.6.2 Library Termination

When the last task in a process exits, or the application’s `main()` routine returns, or any task in the process explicitly calls the POSIX `exit()` routine, the process will be terminated. When a process terminates, the operating system performs the default cleanup actions (see 2.3.1 Real-time Process Life-Cycle, p.12 for information about process termination).

These default actions include: all existing destructors are called in the reverse order of the related constructors, all registered cleanup routines are called in the reverse order of their registration (see Using `atexit()` for Termination Routines, p.31), any remaining process tasks are deleted, the process’ memory pages are returned to the general pool, and any objects created by the process’ tasks in the kernel are deleted (if they are owned by the terminated process).
Additional cleanup work can be stipulated by using `atexit()` or `taskDeleteHookAdd()`, and if pthreads are used, `pthread_cleanup_push()` can be employed.

### Using atexit() for Termination Routines

While there is no library termination routine facility comparable to that for initialization routines (particularly with regard to ranking), a library can register cleanup routines to be executed using the POSIX `atexit()` routine. These routines are executed in reverse order of their registration when a process is being terminated.

Using `atexit()` is useful for cleanup activities that are not handled automatically by the operating system, such as shutting down a device. The call to `atexit()` can be made at anytime during the life of the process, although it is preferably done by the library’s initialization routine.

Cleanup routines registered with `atexit()` are called when `exit()` is called. Note that if a process’ task directly calls the POSIX `_exit()` routine, none of the cleanup routines registered with `atexit()` will be executed.

#### 2.6.3 Developing Static Libraries

The VxWorks development environment provides simple mechanisms for building libraries, including a useful set of default makefile rules. Both Workbench and command line can be used to build libraries.

At the command line, an application library can be compiled for VxWorks by creating a simple makefile and executing a `make` command in the directory that contains the library’s C file. Makefile macros allow for compiling the same source code for different architectures, with different compilers, and so on.

For example, to create a static archive called `libfoobar.a` from various `foo` and `bar` modules, a makefile might look like this:

```makefile
# This file contains make rules for building the foobar library
LIB_BASE_NAME = foobar
OBJJS = foo1.o foo2.o \ 
       bar1.o bar2.o
include $(WIND_USR)/make/rules.library

The makefile includes the default rules defined in `rules.library`. By default, static libraries are created in `installDir/vxworks-6.x/target/usr/lib/arch/CPU/common`.```
The makefile defaults are for the PowerPC architecture and the Wind River Compiler (Diab). Variable assignments can be used at the command line to identify other architectures and compilers. For example, the command used to build the application with the GNU compiler for Pentium2 would be as follows:

```
make CPU=PENTIUM2 TOOL=gnu
```

For more information about using the VxWorks command-line build environment, see the *VxWorks Command-Line Tools User’s Guide: Building Applications and Libraries*.

Also see 2.6.4 Developing Shared Libraries, p.32, and particularly Using the VxWorks CLI Build Environment, p.37, which includes information about creating makefiles to build both static and shared versions of libraries.

For information about using Workbench, see the Wind River Workbench User’s Guide.

### 2.6.4 Developing Shared Libraries

Shared libraries are made up of routines and data that can be used by applications, just like static (archive) libraries. However, when an application is linked to a shared library, the linker does not copy object code from the library into the executable—they are not statically linked. Instead it copies information about the name of the shared library (its *shared object name*) and its run-time location (if the appropriate compiler option is used).

Conceptually, shared libraries do the same job as static libraries. The key differences in their utility are that:

- Statically-linked applications link only those elements of a static library that they need. The entire library does not necessarily become part of the system. If multiple applications in a system use the same library elements, it means that there is replication of library code in the system.

- Dynamically-linked applications require the presence of the entire shared library in the system, even if they only use a small set of its features. If multiple applications in a system need the shared library, however, they share a single copy. The library code is not duplicated.

Applications that make use of shared libraries must be built as dynamic executables, which includes a dynamic linker that carries out the binding of the shared library and application at run time.
Once loaded into memory by the dynamic linker, shared libraries are held in sections of memory (shared data areas) that are accessible to all applications. Each application that uses the shared library gets its own copy of the private data, which is stored in its own memory space. When the last application that references a shared library exits, the library is removed from memory.

Plug-ins (also referred to as dynamically linked libraries, or DLLs) are another class of shared object similar to shared libraries. An application can load or unload a plug-in using a special set of routines. Plug-ins allow for dynamic modification of an application at run time. They can also be used to modify the functionality of a deployed application by loading replacement plug-ins rather than replacing the entire application. (See Using Plug-Ins, p.52.)

**Dynamic Linking**

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

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

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

Because VxWorks uses a flat (that is, not overlapped) virtual memory space, the
NetBSD version of the dynamic linker was modified so that the code and data
segments of a shared library can have different relative addresses at run time. In
standard PIC, the code can locate its GOT by computing the offset from the code’s
load address; overlapped memory allows each private copy of the GOT to have the
same virtual address. With VxWorks PIC, an RTP (process) variable is used instead
to locate a private table of GOTs; each resident shared object reserves a fixed offset
in this GOT table, with which a program can recover the GOT address for a given
shared object.

Advantages and Disadvantages of Shared Libraries

Shared libraries can provide advantages of footprint reduction, flexibility, and
efficiency:

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

- Plug-ins (DLLs) provide flexibility in allowing for dynamic configuration of
  applications.

- Shared libraries are efficient because their code requires fewer relocations than
  standard code when loaded into RAM. Moreover, lazy binding allows for
  linking only those functions that are required.

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

If deferred binding is also used, it introduces non-deterministic behavior.
The startup cost of shared libraries makes up the largest efficiency loss (as is the case on UNIX). It is also greater because of more complex memory setup and more I/O (file accesses) than for static executables.

Shared libraries are therefore most useful when:

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

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

**Additional Considerations**

There are a number of other considerations that may affect whether to use shared libraries:

- Assembly code that refers to global functions or data must be converted by hand into PIC in order to port it to a shared library.
- In shared libraries, the relocation process only affects the data segment. Read-only data identified with the `const` C keyword are therefore gathered with the data segment and not with the text segment to allow a relocation per executable. This means that read-only data used in shared libraries are not protected against erroneous write operations at run-time.
- Code that has not been compiled as PIC will not work in a shared library.
- Code that has been compiled as PIC does not work in an executable program, even if the executable program is dynamic. The dynamic linker does not edit any PIC prologues in the executable program therefore PIC does not work there.
- All constructors in a shared library are executed together, hence a constructor with high priority in one shared library may be executed after a constructor with low priority in a shared library that was loaded later than the first one. All shared library constructors are executed at the priority level of the dynamic linker’s constructor from the point of view of the executable program. (The constructor for the dynamic linker has a priority of 6.)
Shared objects are not cached (they do not linger) if no currently executing program is using them. There is, therefore, extra processor overhead if a shared library is loaded and unloaded frequently.

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

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

### Configuring VxWorks for Shared Libraries

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

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

Doing so automatically includes these components as well:

- `INCLUDE_RTP`, the main component for real-time process support
- `INCLUDE_SHARED_DATA` for storing shared library code
- `INCLUDE_RTP_HOOKS` for shared library initialization
- and various `INCLUDE_SC_XYZ`, the relevant system call components

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

- `INCLUDE_RTP_SHOW`
- `INCLUDE_SHL_SHOW`
- `INCLUDE_SHARED_DATA_SHOW`

Note that if you use the `INCLUDE_SHOW_ROUTINES` component, all of the above are automatically added.

Configuration can be simplified through the use of component bundles. Both `BUNDLE_RTP_DEVELOP` or `BUNDLE_RTP_DEPLOY` provide support for shared libraries (see *Component Bundles*, p.10).
Building Shared Libraries and Dynamic Applications

There are three alternative approaches to building shared libraries and the dynamic applications that make use of them:

- Use Wind River Workbench. All of the build-related elements are created automatically as part of creating and associating shared library and application projects. For information in this regard, see the Wind River Workbench User’s Guide: Shared Library Projects.

- Use the make build rules and macros provided with the VxWorks installation to create the appropriate makefiles, and execute the build from the command line. See Using the VxWorks CLI Build Environment, p.37 and the VxWorks Command-Line Tools User’s Guide: Building Applications and Libraries.

- Write makefiles and rules from scratch, or make use of a custom or proprietary build environment. See Using a Custom Build Environment, p.43, which includes information about the compiler flags used for shared libraries and dynamic applications.

Using the VxWorks CLI Build Environment

Using the command-line interface (CLI) build environment provided with the VxWorks installation allows you to take advantage of a set of default makefile rules to build dynamic executables and shared libraries.

It can be useful to set up a project with a top-level directory with subdirectories for the application and for each shared library. The makefiles would be:

- A top-level makefile that identifies the application and library subdirectories to build.
- A makefile for each shared library.
- A makefile for each application.

The directory structure might look like the one in Figure 2-1. The grey boxes indicate directories created automatically by the default build rules.
If you do not use a top-level makefile for the whole project, and you build libraries and applications separately, you must of course build libraries first.

**Top-Level Makefile**

The top level makefile requires only two elements. For example:

```
SUBDIRS = libA libB app
include $(WIND_USR)/make/rules.rtp
```

The `SUBDIRS` variable indicates what subdirectories to build. In this case, there are two library subdirectories (`libA` and `libB`) and an application subdirectory (`app`) identified.

The include statement references `rules.rtp`, which is the make fragment holding build rules for application (.vxe) files.

The `$WIND_USR` element is an environment variable pointing to the user-side of the target tree: `installDir/vxworks-6.x/target/usr`.

**Library Makefile**

Only four elements are required for a library makefile. For example:

```
LIB_BASE_NAME = MyFoo
OBJS = foo1.o foo2.o foo3.o
EXTRA_INCLUDE += -I/home/moimoi/proj/h
include $(WIND_USR)/make/rules.library
```

The `LIB_BASE_NAME` variable specifies the library’s *basename*. It takes a *stripped* version of the library name; that is, without the `lib` prefix and without the filename extension. The `LIB_BASE_NAME` variable is required for generating the shared object name (soname) information, which is necessary for identifying the run-time
location of the shared library. See *Shared Object Names, Library Versions, and Run-time Locations*, p. 48.

The **OBJ**S variable identifies the object files that make up the library.

The **EXTRA_INCLUDE** variable appends additional search paths for the include files (using the `+=` operator).

The include statement references **rules.library**, which is the make fragment holding build rules for libraries.

The `$WIND_USR` element is an environment variable pointing to the user-side of the target tree: `installDir/vxworks-6.x/target/usr`.

Additional makefile macros and conditional statements can be used to build either shared or static libraries, to specify a location other than the default in which to create the library, and so on. A more useful makefile would look like this:

```makefile
SL_INSTALL_DIR = /home/moimoi/proj/ppc32/lib
ifeq ($(EXE_FORMAT),dynamic)
  LIB_FORMAT = shared
endif
ifeq ($(LIB_FORMAT),shared)
  LOCAL_CLEAN += $(OBJ_DIR)/*.sho $(OBJ_DIR)/libMyFoo.so
else
  LOCAL_CLEAN += $(OBJ_DIR)/*.o $(OBJ_DIR)/libMyFoo.a
endif
LIB_BASE_NAME = MyFoo
LIBNAME = lib$LIB_BASE_NAME
SL_VERSION = 2
OBJ = module1.o module2.o
OBJ_DIR = $(CPU)$TOOL
EXTRA_INCLUDE += -I/home/moimoi/proj/h
include $(WIND_USR)/make/rules.library
LIBDIR = $(OBJ_DIR)
```

The **SL_INSTALL_DIR** variable specifies a non-default location for the library file. It is often useful to keep project work outside of the installation directory. Note that if you are generating code for different processor architectures, you could use the `$CPU` variable to define the architecture-specific subdirectory (which would then use the Wind River naming conventions—**PPC32** instead of **ppc32**, as in this case).

The **LIB_FORMAT** variable specifies whether the library must be generated as a static library (the default if not specified) or a shared library. It can be set to static, shared, or both. See *Library Build Commands*, p. 40.

The **LOCAL_CLEAN** variable specifies what must be deleted when non-default directories are used.

The **OBJ_DIR** variable specifies a non-default directory in which the object modules are generated.
The **LIBNAME** variable specifies a non-default directory for the static library.

The **SL_VERSION** variable is used to create a version number for a shared library. By default, the shared library version number is one (libName.so.1), so the variable is not needed unless you want to build an alternate version of the shared library. For more information about shared library versions, see *Shared Object Names, Library Versions, and Run-time Locations*, p.48.

The **LIBDIR** variable prevents the creation of an unused directory in installDir/vxworks-6.x/target/usr/lib when set to $(OBJ_DIR), as it is in this example. This variable must be the last one used; that is, it must be listed after the inclusion of rules.library in order to take precedence.

**Library Build Output**

By default, a library is created as a static archive file (libName.a) in installDir/target/usr/lib/arch/cpu/common. For example, installDir/target/usr/lib/ppc/PPC32/common.

When a library is built as a shared library:

- The reference file (libName.so) is created by default in the same default directory as a static library file. The reference file is used by the linker when a dynamic application is built.

- The runtime version of the shared library (libName.so.n) is created by default in the same default directory as the application executable.

That is, all the runtime executables are created in the same binary directory by default: installDir/target/usr/root/cpuTool/bin. For example, installDir/target/usr/root/PPC32diab/bin.

As noted in *Library Makefile*, p.38, the **SL_INSTALL_DIR** makefile variable can be used to specify a non-default location for the library file.

**NOTE:** The rather cumbersome usage of **LIBNAME**, **OBJ_DIR** and **LIBDIR** is required by the library rules. The three must be specified if one does not want to clutter the distribution’s installDir/vxworks-6.x/target/usr/lib/arch/cpu/common directory with unnecessary directories and user application-specific files.

**Library Build Commands**

The first conditional statement in the example above allows for building both a shared library and dynamic application using the top-level makefile. For example:

`make EXE_FORMAT=dynamic`
builds the dynamic executable and related shared libraries for the PPC32 architecture with the Wind River Compiler (the default architecture and compiler).

The following command is similar, but uses the GNU compiler to build for the Pentium architecture:

```
make CPU=PPC32 TOOL=gnu EXE_FORMAT=dynamic
```

Static libraries are created by default, so the `EXE_FORMAT` variable need not be used to create them.

You can also build libraries by executing `make` in the directory containing the library makefile. The `LIB_FORMAT` variable allows you to create static, shared, or both sets of libraries. For example:

```
make LIB_FORMAT=static
make LIB_FORMAT=shared
make LIB_FORMAT=both
```

Static libraries are created by default, so that option is superfluous in this context.

### Application Makefile

Only four elements are required for an application makefile when the application is a dynamic executable that uses a shared library. For example:

```make
EXE = myVxApp.vxe
OBJS = myVxApp.o
ADDED_LIBS += -L../lib/$(CPU)$(TOOL) –lMyFoo
include $(WIND_USR)/make/rules.rtp
```

The `EXE` variable defines the name of the application executable.

The `OBJS` variable identifies the constituent object modules. If more than one is used, they are listed in a space-separated string. For example:

```make
OBJS = bar1.o bar2.o bar3.o
```

The `ADDED_LIBS` variable appends the search paths and stripped names of any custom libraries that the application requires (with the `+=` operator). This option can be used for creating either static or dynamic applications. See *Compiler Flag For Shared Objects and Dynamic Applications*, p.45 for a discussion of the `-l` flag.

The include statement and `$WIND_USR` element are described in *Top-Level Makefile*, p.38.

Additional makefile macros can (and often should) be used to specify a different location for the application executable, the runtime location of the shared libraries upon which it depends, conditional compilation of either a dynamic or static application, and so on. A more useful makefile would look like the following:
The `VXE_DIR` variable identifies an alternative to the default directory in which the application executable is created. It is often useful to keep project work outside of the installation tree.

The `LIB_DIR` variable is simply a local make variable that can be used conveniently to identify where a library is located (if it is not in the default location) in the `ADDED_LIBS` and `ADDED_LIB_DEPS` lines without repeating the literal path information.

The `ADDED_DYN_EXE_FLAGS` variable is used to pass additional compiler flags specific to the generation of dynamic executables. It is used here with the `-Wl,-rpath` flag to specify the run-time location of any shared libraries upon which the application depends. In this example, the location is in a `lib` subdirectory of the ROMFS file system on the VxWorks target:

```
ADDED_DYN_EXE_FLAGS    += -Wl,-rpath /romfs/lib
```

In this next example, the location is on the host system:

```
# ADDED_DYN_EXE_FLAGS    += -Wl,-rpath c:/myProj/lib/SIMPENTIUMdiab
```

For information about the `-Wl,-rpath` flag, see Compiler Flag For Shared Objects and Dynamic Applications, p. 45. Also see Shared Object Names, Library Versions, and Run-time Locations, p. 48.

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

The `ADDED_LIB_DEPS` specifies the dependencies between the application and the application's static libraries (with the `+=` operator). For static linkage, this variable forces a rebuild of the application if the static library has been changed since the last build.
2.6 Developing Application Libraries

Application Build Output

By default, the executable is created in `installDir/target/usr/root/cpuTool/bin`, as are the run-time shared object files (libraries and plug-ins). For example, `installDir/target/usr/root/PPC32diab/bin`.

As noted in Application Makefile, p. 41, the VXE_DIR variable can be used to identify an alternative output directory.

Application Build Commands

In the example provided in Application Makefile, p. 41, the conditional `ifeq/else/endif` statement allows for generating either a static executable or a dynamic executable, depending on the value given the EXE_FORMAT variable when make is executed at the command line. The EXE_FORMAT variable can take one of two values, dynamic or static. For example:

```plaintext
make EXE_FORMAT=dynamic
```

As a static build is the default, setting EXE_FORMAT to static is not necessary.

Using a Custom Build Environment

If you do not use Wind River Workbench or the CLI VxWorks build environment to create shared libraries and dynamic executables, follow the guidelines provided in this section (including the compiler flags listed in Compiler Flag For Shared Objects and Dynamic Applications, p. 45).

To create a dynamically linked program or shared object, you must run the static linker (`dld` or `ld`) with specific flags. These flags cause the linker to mark the output as dynamically linked, and to include various data structures for later dynamic linking.

Shared objects are compiled as position-independent code (PIC). Note that all code that you compile for a shared object must be PIC. Do not use PIC options to compile modules that will be statically linked into an executable program.

A shared library must be built before any dynamic application that makes use of it, in two steps:

1. Generate the PIC object modules that are to be assembled into a shared object file. The modules are created with the `.sho` filename extension.
2. Generate the shared object (with the `.so` filename extension) based on the PIC object modules.

The dynamic application can then be built, in two more steps:
1. Generate the object modules that will be assembled into an executable file. The application’s object modules are created with a .o filename extension.

2. Generate the executable file based on the object modules. By convention, the application is created with the .vxe filename extension.

**Generating PIC Object Modules**

The following examples illustrate compiler commands used to generate a library’s PIC modules. The compiler flags are the same as those used by the default makefile system.

Using the Wind River Compiler:

```
```

Using the GNU Compiler:

```
```

⚠️ **CAUTION:** The combination of PIC modules generated by the Wind River Compiler and the GNU compiler is not supported and is not expected to work. Doing so may lead to unpredictable results.

**Generating the Shared Library**

The following examples illustrate compiler commands used to generate a library from PIC modules. The compiler flags are the same as those used by the default makefile system.

Using the Wind River Compiler:

```
```
Using the GNU Compiler:


Generating the Application’s Object Modules

The following examples illustrate compiler commands used to generate an application’s object modules. The compiler flags are the same as those used by the default makefile system.

Using the Wind River Compiler:

dcc -tPPCEH:rtp -Xansi -XO -I$(WIND_BASE)/target/usr/h -I/home/moimoi/proj/h -DCPU=PPC32 -DTTOOL_FAMILY=diab -DTTOOL=diab -c main.c -o PPC32diab/main.o

Using the GNU Compiler:


Generating the Application Executable

The following examples illustrate compiler commands used to generate the application executable. The compiler flags are the same as those used by the default makefile system.

Using the Wind River Compiler:


Using the GNU Compiler:


Compiler Flag For Shared Objects and Dynamic Applications

The most important compiler flags used in generating shared objects (shared libraries and plug-ins) and dynamic applications are described below. Note that no specific flags are required for generating the application modules (the .o files), but
that special flags are required for generating an application executable (the .vxe file).

**Flags for Shared Objects**

To generate position-independent code (PIC), suitable for use in a shared library, use:

- `-Xpic` with the Wind River Compiler
- `-fpic` with the GNU Compiler

To link shared library PIC modules, use:

- `-Xshared` with the Wind River Compiler
- `-shared` with the GNU Compiler

To name a shared object (and optionally provide a version number), use:

- `-soname=sharedLibraryName` with the Wind River Compiler
- `-Wl,-soname,sharedLibraryName` with the GNU Compiler

Note that `gcc` does not recognize `-soname` so it must be passed to `ld` with `-Wl`.

The `-soname` flag specifies a shared object name for a shared library. This information is used both for locating shared libraries at run-time, as well as for creating different versions of shared libraries. See *Defining Shared Object Names and Shared Library Versions*, p.50

To force the object to use its own definitions for global functions and data even if some other shared object, or the dynamic executable, would otherwise override those definitions, use:

- `-Bsymbolic` with either compiler

  Note that the compilers do not recognize `-Bsymbolic`, so it must be passed to their linker with `-Wl`.

Normally when the dynamic linker looks for a definition for a function or some data, it chooses the first one that it sees, even if some later module has its own definition.

**Flags for Application Executables**

To allow an application to dynamically link with a shared library, use:

- `-Xdynamic` with the Wind River Compiler
- `-non-static` with the GNU Compiler
To identify the runtime path to shared libraries, use the following when building the application’s dynamic executable:

- **-Wl,-rpath** `sharedLibDirPath` with either compiler

Provide a semicolon-separated list of directory names (the run path), which the dynamic linker uses as one of the means to locate shared libraries at runtime. See *Shared Object Names, Library Versions, and Run-time Locations*, p. 48 for more information in this regard. The runtime dynamic linker can only make use of this information if the shared library is created with a shared object name (see the -soname flag description in *Flags for Shared Objects*, p. 46).

⚠️ **CAUTION:** The dynamic linker assumes that library files with the same name are different if the path defined with -rpath is different. Care should therefore be taken to ensure that multiple copies of identical libraries are not loaded.

The paths provided with the -Wl,-rpath linker flag must, of course, be accessible to the VxWorks target system. If the paths reference files on the host’s file system, make sure that they are identified in a way that is appropriate for the host-target connection (NFS is transparent; FTP requires `hostname:` before the path if it is not on the same system from which VxWorks was booted; the VxWorks simulator requires a `host:` prefix; and so on).

For example, the following example identifies a **lib** subdirectory in the ROMFS file system in VxWorks, the c:\proj\lib on a Windows host system, and `/home/moimoi/proj` on an NFS file system as the locations for shared libraries:

```
-Wl,-rpath /romfs/lib;c:/proj/lib;/home/moimoi/proj
```

When using Wind River Workbench to create a dynamic executable, the linker option for a dynamic executable must be selected. The -Wl,-rpath option can be used to identify shared library locations, if necessary.

To indicate which shared libraries will be used by the dynamic application, use:

- **-l** with either compiler

  The -l flag takes a stripped version of the library name. By convention, libraries referred to by the -l flag are named `libname.so` or `libname.a`. Both the lib part of the name and the filename extension (.so or .a) are omitted when specified with -l. For example, both `libMyFoo.so` and `libMyFoo.a` could be identified with -l MyFoo. This facility is particularly useful for conditional compilation of either static or dynamic applications (see *Application Makefile*, p. 41).
To have the dynamic linker to look up and bind functions only when the function is first called, use:

- `-Xbind-lazy` with either compiler

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

Note that you can also select or disable lazy function binding using environment variables when you start an application. If the variable `LD_BIND_NOW` is non-null, the dynamic linker uses immediate binding. If the variable `LD_BIND_LAZY` is non-null, then the dynamic linker uses lazy binding.

**Shared Object Names, Library Versions, and Run-time Locations**

Shared libraries can be created with shared object names. These names can be used to identify different versions of shared libraries. They can also be used in conjunction with a variety of mechanisms to identify the runtime locations of shared libraries, so that the application’s dynamic linker can find them.

A shared object name is defined when the library is built, and is incorporated into the executable file. It is independent of the library’s run-time filename, but is often the same. A shared object name is often referred to as an `soname` (after the `-soname` compiler flag), and is effectively the run-time name of the shared library.

When a shared library is built with a shared object name, the name is stored in an ELF `SONAME` record. When a dynamic application is linked against that shared library, the `soname` information is recorded in an ELF `NEEDED` record for use at run-time by the dynamic linker.

**Locating Shared Libraries at Run-time**

If a shared library is created without `soname` information, the application’s dynamic linker only looks for the run-time shared library file in same directory as the one in which the application executable is located.

By default, the VxWorks makefile system creates both the application executable and run-time shared library files in the same directory (`Application Makefile`, p.41), which facilitates running the application during the development process. For a deployed system, shared libraries created without version information will work
only if they are stored in the same directory as the application that requires them (for example, if they are both stored in the same ROMFS directory). However, if the dynamic library files are stored somewhere else on the host or target system, as will often be the case, the application’s dynamic linker will not be able to find them.

If a shared library is created with shared object name information, the dynamic linker can use environment variables, configuration files, compiled location information, and a default location to find the shared libraries required by its applications. The linker checks the directories provided by the following mechanisms, in this order:

1. The environment variable `LD_LIBRARY_PATH`, which can be set to a semicolon-separated list of directories. (See below for more information.)

2. The configuration file `ld.so.conf`. By default the dynamic linker looks for this file in the same directory as the one in which the application executable resides. The location can also be specified with the `LD_SO_CONF` environment variable. The `ld.so.conf` file simply lists paths, one per line with the pound sign (#) at the left margin for comment lines.

3. The run paths identified at build time with the `–rpath` option.

4. The same directory as the application file itself.

In addition, the `LD_PRELOAD` environment variable can be used to identify a set of libraries to load at startup time, before loading any other shared libraries. The variable can be set to a semicolon-separated list of library files. For example:

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

The `LD_LIBRARY_PATH` environment variable can be used when the application is started. Using the shell’s command interpreter, for example, the syntax would be as follows:

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

Note in particular that there are no spaces within the quote-enclosed string, but that there is a comma separating the paths in the quoted string; and that there is one space between the string and the executable argument to `rtp exec`.

If, for example, the application and shared libraries were stored in ROMFS on the target in `app` and `lib` subdirectories, the command would look quite tidy:

```
rtp exec -e "LD_LIBRARY_PATH=/romfs/lib" /romfs/app/myVxApp.vxe one two three
```
In this next example, the command (which would be typed all on one line, of course), identifies the location of libc.so.1 as well as a custom shared library on the host system:

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

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

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

**Using Different Versions of Shared Libraries**

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

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

**Defining Shared Object Names and Shared Library Versions**

Shared object name and version information can be defined with the following:

- The `SHARED_LIB_VERSION` build macro for Wind River Workbench.
- The `LIB_BASE_NAME` and `SL_VERSION` make macros for the default build system (see `Library Makefile`, p.38).
- The compiler’s `-soname` flag (see `Compiler Flag For Shared Objects and Dynamic Applications`, p.45).
By default, the VxWorks CLI build environment create version one instances of dynamic shared libraries (that is, libName.so.1). With Wind River Workbench, the user must set the SHAREDLIB_VERSION build macro explicitly.

VxWorks Run-time C Library libc.so

The VxWorks distribution provides a run-time shared library that is similar to the UNIX C run-time library. The VxWorks shared library libc.so includes all of the user-side libraries except for the following:

- `aioPxLib` (see 6.6 Asynchronous Input/Output, p.277)
- `memEdrLib` (see 5.4 Memory Error Detection, p.248)
- message channel libraries (see 3.3.8 Message Channels, p.133)
- networking libraries (see the Wind River Network Stack for VxWorks 6 Programmer’s Guide)

The libc.so shared library provides all of the basic facilities that an executable might need. It is the shared library equivalent of libc.a. All dynamic executables require libc.so.1 at run time.

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

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

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

installDir/vxworks-6.x/target/usr/root/SIMPENTIUMdiab/bin/libc.so.1

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

For deployed systems, the libc.so.1 file can be copied manually to whatever location is appropriate. The most convenient way to make the dynamic linker
aware of the location of libc.so.1 is to store the file in the same location as the dynamic application, or to use the -Wl,-rpath compiler flag when the application is built. See Locating Shared Libraries at Run-time, p.48 for more information.

Using Plug-Ins

A plug-in is a shared object that can be loaded by an application at run-time to modify or add functionality. It is functionally equivalent to a dynamically linked library (DLL) in other operating systems. A plug-in is built in exactly the same way as a shared library, except that it does not need a shared object name.

Applications that use plug-ins must include the dlfcn.h header file (as in the example provided below). They must also be compiled as dynamic executables, even if they do not use shared libraries. Static executables cannot load plug-ins because they do not have a dynamic symbol table.

The location of the plug-in is coded into the application itself. The application makes API calls to the dynamic linker to load the plug-in and to access its functions and data. The libdl library, which provides the APIs for these calls, is automatically linked into a dynamic executable.

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

```c
#include <dlfcn.h>

typedef void *PROTO;
const char plugin_path[] = "/romfs/plug-ins";

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

    n = sizeof plugin_path + 1 + strlen(name) + 3;
    if ((path = malloc(n)) == -1) {
        fprintf(stderr, "can't allocate memory: %s",
                strerror(errno));
        return NULL;
    }
    sprintf(path, "%s/%s.so", plugin_path, name);
    if ((handle = dlopen(path, RTLD_NOW)) == NULL)
```
Assume you implement a new protocol named `reliable`. You would compile the code as PIC, then link it using the `-Xdynamic -Xshared` flags (with the Wind River compiler) into a shared object named `reliable.so` (the comparable GNU flags would be `-non-static` and `-shared`). You install `reliable.so` as `/romfs/plug-ins/reliable.so` on the target.

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

If a plug-in must make calls into the application, it may be necessary to link the application with the `-E` flag (with either the Wind River Compiler or the GNU compiler) to force the dynamic linker to register all the necessary symbols for the plug-ins before they are loaded. Otherwise, the plug-in may not have access to all the symbols to which it should link. The only case in which you would not need to use the `-E` flag is when the process’ shared libraries require all of the same symbols as the plug-in.
Dynamic Linking Routines for Plug-Ins

The routines used with plug-ins are:

\[ \text{void *} \text{dlopen(const char *path, int mode)} \]
Load the shared object from the given pathname.

\[ \text{void *} \text{dlsym(void *handle, const char *name)} \]
Look up the function or data identified by \text{name} in the shared object that is described by \text{handle}, and return its address.

\[ \text{int dlclose(void *handle)} \]
Remove a reference to the shared object that is described by \text{handle}. If this is the last reference, \text{dlclose( )} unloads the shared object.

\[ \text{char *} \text{dlerror(void)} \]
Return the error string after an error in \text{dlopen( ), dlclose( ) or dlsym( )}.

For more information, see the \text{libdl} API references.

Using readelf to Examine Dynamic ELF Files

The \text{readelf} tool can be used to extract dynamic records from an executable or a shared object, such as a shared object name and path.

Use the version that is appropriate for the target architecture; for example, use \text{readelfppc} on a PowerPC file. The various versions of the tool are provided in \text{installDir}/gnu/3.3.2-vxworks6x/hostType/bin.

The -d flag causes \text{readelf} to list dynamic records by tag type, such as:

\text{NEEDED}
A required shared library. There is one \text{NEEDED} record for every library that a given dynamic ELF file depends on. The \text{NEEDED} record instructs the dynamic linker to load the given library at run time, in order to include definitions that are needed by the given ELF file. Both dynamic executable programs and shared objects may use \text{NEEDED} records. The dynamic linker loads shared libraries in the order in which it encounters \text{NEEDED} records. (It is useful to know that the dynamic linker executes the constructors in each shared library in reverse order of loading.)

\text{SONAME}
The shared object name.

\text{RPATH}
The run path.
Getting Runtime Information About Shared Libraries

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

The two commands below starts the `tmPthreadLib.vxe` application in the background, so that the shell is available for other commands:

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

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

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

```bash
[vxWorks *]# rtp

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

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

```bash
[vxWorks *]# shl

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

Note that the `shl info` command will provide the full path.
The *sd* command provides information about shared data regions. In this case the regions are used by the two shared libraries (one region for each shared library and one for the GOT).

```
[vxWorks *]# sd
NAME         ID     VIRT ADDR      PHYS ADDR         SIZE    CLIENT CNT
------------ ---------- ---------- ------------------ ---------- ----------
/dl-text-s >   0x95e158 0x10031000          0x183b000     0xa000          1
/dl-text-s >   0x95e428 0x10042000          0x18e4000     0x1000          1
/dl-text-s >   0x95e1d0 0x10043000          0x184c000    0x60000          1
```

You can get information about a specific shared data region with the *sd info* command. In this case, information is displayed about the region with ID 0x95e158.

Note that in the **dl-text-segments** part of the full name is the convention for shared data. The forward slash indicates that it is public. And this shared data region holds the shared library **libPthreadLibTest.so.1**.

```
[vxWorks *]# sd info 0x95e158
NAME         ID     VIRT ADDR      PHYS ADDR         SIZE    CLIENT CNT
------------ ---------- ---------- ------------------ ---------- ----------
/dl-text-s >   0x95e158 0x10031000          0x183b000     0xa000          1
```

Full Name: /dl-text-segments./libPthreadLibTest.so.1

Default MMU Attributes (0x87f):

```
ACCESS                  CACHE
----------------------- ---------
RWX / RWX  DEFAULT  --
```

Clients:

```
NAME         ID          ACCESS                  CACHE
------------ ----------  ----------------------- ----------
./tmPthrea >   0x80a4f0  R-X / R-X   CB-/--/-  --
```

In this next example, the application is started in stopped mode. The job number is 1.

```
[vxWorks *]# rtp exec -s tmPthreadLib.vxe 2 &
Launching process 'tmPthreadLib.vxe' ...
Process 'tmPthreadLib.vxe' (process Id = 0x808510) launched.
Attachment number for process 'tmPthreadLib.vxe' is $1.
```

Here, the job number is used (with the ampersand sign) to switch to the process address space, as indicated by the prompt. A breakpoint is set on **main()** with **bp**, and execution is continued with **rtpc**.
Then a new breakpoint is set, all breakpoints listed, and execution continued.

![Breakpoint Address Ctx Ctx Id Cnt Stops N Hard](image)

The `l` alias for `mem list` provides disassembly from address 0x10038e34, and one instruction is executed using the `s` alias for `task step`.

![Disassembly](image)
Debugging Shared Libraries

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

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

When a shared library cannot be found, make sure that its location has been correctly identified or that it resides in the same location as the executable (Locating Shared Libraries at Run-time, p. 48). If the shared libraries are not stored on the target, also make sure that they are accessible from the target.

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

Note that shared library symbols are not visible if an application is started in stopped mode. Until execution of _start() (the entry point of an application provided by the compiler) calls the dynamic linker, shared library symbols are not yet registered. (For information about symbol registration, see 2.8.3 Applications and Symbol Registration, p. 73.)

Working With Shared Libraries From a Windows Host

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

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

To make use of NFS, you can either install an NFS server on Windows or make use of remote access to a UNIX machine that runs an NFS server. If you have remote access, you can use the UNIX machine to boot your target and export its file system. In this case you must set up your Workspace with a VxWorks File System Project on the remote UNIX machine’s file system, which in turn exports it to the target.

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

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

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

1. In Windows Explorer, select your Workspace and use the context menu to select **Share…**
2. Select the **NFS Sharing** tab.
3. Enter **Share this folder**, Share name = **Workspace**
4. Enable **Allow anonymous access**. This provides the VxWorks target with read-only access to the share without having to set up user mappings or access permissions.

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

Adding the **INCLUDE_NFS_MOUNT_ALL** component provides all the necessary features. (Using Wind River Workbench, the **VxWorks Image Configuration Editor** can be used to add the component from the following node: **Network Components > Network Applications > NFS Components > NFS mount all**.) Make sure the target the target connection is disconnected before you rebuild your kernel image.
When you reboot the target it automatically mounts all NFS shares exported by the host. To verify that VxWorks can access your NFS mount, use the `devs` and `ls "/Workspace"` commands from the kernel shell.

### 2.7 Creating and Managing Shared Data Regions

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

The shared data region facility provides no inherent facility for mutual exclusion. Applications must use standard mutual exclusion mechanisms—such as public semaphores—to ensure controlled access to a shared data region resources (see *3.3 Intertask and Interprocess Communications*, p.105).

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

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

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

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

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

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

Operations on shared data regions are not restricted to applications—kernel tasks may also perform these operations.
Shared data regions use memory resources from both the kernel’s and the application’s memory space. The kernel’s heap is used to allocate the shared data object. The physical memory for the shared data region is allocated from the global physical page pool.

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

Shared data regions can be created in systems with and without MMU support. Also see 3.3.2 Shared Data Structures, p.107 and 4.18.3 Shared Memory Objects, p.236

### 2.7.1 Configuring VxWorks for Shared Data Regions

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

### 2.7.2 Creating Shared Data Regions

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

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

The creation routines take parameters that define the name of the region, its size and physical address, MMU attributes, and two options that govern the regions persistence and availability to other processes.
The MMU attribute options define access permissions and the cache option for the process' page manager:

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

By default, the creator process always gets read and write permissions for the region, regardless of the permissions set with the creation call, which affect all client processes. The creator, can however, change its own permissions with \texttt{sdProtect()}. See \textit{Changing Shared Data Region Protection Attributes}, p.62.

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

### 2.7.3 Accessing Shared Data Regions

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

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

Mutual exclusion mechanisms should be used to ensure that only one application can access the same shared data region at a time. The \texttt{sdLib} library does not provide any mechanisms for doing so automatically. For more information about mutual exclusion, see \textit{3.3 Intertask and Interprocess Communications}, p.105.

### Changing Shared Data Region Protection Attributes

The MMU attributes of a shared data region can be changed with \texttt{sdProtect()}. The change can only be to a sub-set of the attributes defined when the region was created. For example, if a region was created with only read and write permissions, these can only be changed to read-only and no access, and not expanded to other permissions. In addition, the changes are made only for the caller’s process; they do not affect the permissions of other processes.
A set of macros is provided with the library for common sets of MMU attribute combinations.

### 2.7.4 Deleting Shared Data Regions

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

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

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

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

### 2.8 Executing Applications

Because a process is an instance of a program in execution, starting and terminating an application involves creating and deleting a process. An RTP application can be started and terminated interactively, programmatically, and automatically with various facilities that act on processes.

An application can be started by:

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

An application can be stopped by the same means as those that terminate processes—a process is an instance of an application in execution after all. See 2.3.1 Real-time Process Life-Cycle, p.12 for information about all the ways in which processes can be terminated.
Application executables can be stored in the VxWorks ROMFS file system on the target system, on the host development system, or on any other file system accessible to the target system (another workstation on a network, for example).

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

2.8.1 Running Applications Interactively

Running applications interactively is obviously most desirable for the development environment, but it can also be used to run special applications on deployed systems that are otherwise not run as part of normal system operation (for diagnostic purposes, for example). In the latter case, it might be advantageous to store auxiliary applications in ROMFS; see 2.9 Bundling Applications with a VxWorks System using ROMFS, p.74.

Starting Applications

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

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

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

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

Note that some types of connections between the target and host require modifiers to the pathname (NFS is transparent; FTP requires $hostname$ before the path if it is not on the same system from which VxWorks was booted; the VxWorks simulator requires a $host$ prefix; and so on).
Using the shell’s command interpreter, the application can be started in two different ways, either directly specifying the path and name of the executable file and the arguments (like with a UNIX shell):

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

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

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

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

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

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

### Stopping Applications

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

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

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

For a description of all the ways in which a process can be terminated, see 2.3.1 *Real-time Process Life-Cycle*, p.12.

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

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

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

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

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

In this section, use of the startup facility is illustrated with applications that reside on the host system. For information about using ROMFS to bundle applications with the operating system, and for examples illustrating how applications in the ROMFS file system are identified for the startup facility, see 2.9 Bundling Applications with a VxWorks System using ROMFS, p.74.

Startup Facility Options

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

- an application startup configuration parameter
- a boot loader parameter
2 Applications and Processes
2.8 Executing Applications

- a VxWorks shell script
- the **usrAppInit()** routine

The components that support this functionality are, respectively:

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

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

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

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

### Application Startup String Syntax

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

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

This syntax involves only two special characters:

#

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

^

A caret delimits individual arguments (if any) to the application. A caret is not required after the final argument.
The carets are not required—spaces can be used instead—with the startup configuration parameter, but carets must be used with the boot loader parameter.

The following examples illustrate basic syntax usage:

```
#c:/apps/myVxApp.vxe
   Starts c:/apps/myVxApp.vxe
#c:/apps/myVxApp.vxe^one^two^three
   Starts c:/apps/myVxApp.vxe with the arguments one, two, three.
#c:/apps/myOtherVxApp.vxe
   Starts c:/apps/myOtherVxApp.vxe without any arguments.
#c:/apps/myVxApp.vxe^one^two^three#c:/apps/myOtherVxApp.vxe
   Starts both applications, the first one with its three arguments.
```

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

- `%p=value`  
  Sets the priority of the initial task of the process. Priorities can be in the range of 0-255.

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

- `%o=value`  
  Sets the process options parameter.

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

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

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

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

If the `rtpSpawn()` options are not set, the following defaults apply: the initial task priority is 220; the initial task stack size is 64 Kb; the options value is zero; and the initial task option is `VX_FP_TASK`. 
The maximum size of the string used in the assignment is 160 bytes, inclusive of names, parameters, and delimiters. No spaces can be used in the assignment, so application files should not be put in host directories for which the path includes spaces.

Specifying Applications with a Startup Configuration Parameter

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

The identification string must use the syntax described in Application Startup String Syntax, p.67. And the operating system must be rebuilt thereafter.

Specifying Applications with a Boot Loader Parameter

The VxWorks boot loader includes a parameter—the `s` parameter—that can be used to identify applications that should be started automatically at boot time, as well as to identify shell scripts to be executed.³ (For information about the boot loader, see the VxWorks Kernel Programmer’s Guide: Kernel.)

Applications can be specified both interactively and statically with the `s` parameter. In either case, the parameter is set to the path and name of one or more executables and their arguments (if any), as well as to the applications’ process-spawn parameters (optionally). The special syntax described above is used to describe the applications (see Application Startup String Syntax, p.67).

This functionality is provided with the `INCLUDE_RTP_APPL_BOOTLINE` component.

Note that the boot loader `s` parameter serves a dual purpose: to dispatch script file names to the shell, and to dispatch application startup strings to the startup facility. Script files used with the `s` parameter can only contain C interpreter commands; they cannot include startup facility syntax (also see Specifying Applications with a VxWorks Shell Script, p.70).

If the boot parameter is used to identify a startup script to be run at boot time as well as applications, it must be listed before any applications. For example, to run

³. In versions of VxWorks 5.x, the boot loader `s` parameter was used solely to specify a shell script.
the startup script file `myScript` and `myVxApp.vxe` (with three arguments), the following sequence would be required:

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

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

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

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

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

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

### Specifying Applications with a VxWorks Shell Script

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

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

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

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

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

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

If a shell script is written for the C interpreter, it can be identified interactively using the boot loader `s` parameter—in a manner similar to applications—using a
sub-set of the same string syntax. A shell script for the C interpreter can also be identified statically with the `DEFAULT_BOOT_LINE` parameter of the `INCLUDE_RTP_APPL_BOOTLINE` component. (See *Specifying Applications with a Boot Loader Parameter*, p.69 and *Application Startup String Syntax*, p.67.)

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

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

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

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

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

Note that shell scripts can be stored in ROMFS for use in deployed systems (see *2.9 Bundling Applications with a VxWorks System using ROMFS*, p.74).
Specify Applications with `usrRtpAppInit()`

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

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

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

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

    /* set the application's arguments */
    argv[0] = vxeName;
    argv[1] = "first";
    argv[2] = "second";
    argv[3] = "third";
    argv[4] = NULL;

    /* Spawn the RTP. No environment variables are passed */
    if ((rtpId = rtpSpawn (vxeName, argv, NULL, 220, 0x10000, 0)) == NULL)
    {
        printErr ("Impossible to start myVxApp application (errno = %#x)",
                  errno);
    }
}
```

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

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

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

For information about bundling applications with the system image in ROMFS, see 2.9 Bundling Applications with a VxWorks System using ROMFS, p.74.
2.8.3 Applications and Symbol Registration

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

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

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

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

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

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

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

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

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

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

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

Note that symbols should not be stripped from executable files (.vxe files) because they are relocatable. And while symbols may be stripped from run-time shared library files (.so files), it makes debugging them more difficult.
2.9 Bundling Applications with a VxWorks System using ROMFS

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

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

When the system boots, the ROMFS file system and the application executables are loaded with the kernel. Applications and operating system can therefore be deployed as a single unit. And coupled with an automated startup facility (see 2.8 Executing Applications, p. 63), ROMFS provides the ability to create fully autonomous, multi-process systems.

This section provides information about using ROMFS to store process-based applications with the VxWorks operating system in a single system image. For general information about ROMFS, see 7.8 Read-Only Memory File System: ROMFS, p. 323.

2.9.1 Configuring VxWorks with ROMFS

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

2.9.2 Building a System With ROMFS and Applications

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

cd c:\myInstallDir\vxworks-6.1\target\proj\wsbc8260_diab
mkdir romfs

```plaintext
copy c:\myInstallDir\vxworks-6.1\target\usr\root\PPC32diab\bin\myVxApp.vxe romfs
```
make TOOL=diab

The contents of the `romfs` directory are automatically built into a ROMFS file system and combined with the VxWorks image.

The ROMFS directory does not need to be created in the VxWorks project directory. It can also be created in any location on (or accessible from) the host system, and the `make ROMFS_DIR` macro used to identify where it is in the build command. For example:

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

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

### 2.9.3 Accessing Files in ROMFS

At run time, the ROMFS file system is accessed as `/romfs`. The content of the ROMFS directory can be browsed using the traditional `ls` and `cd` shell commands, and accessed programmatically with standard file system routines, such as `open()` and `read()`.

For example, if the directory

```
installDir/vxworks-6.x/target/proj/wrSbc8260_diab/romfs
```

has been created on the host, `myVxApp.vxe` copied to it, and the system rebuilt and booted, then using `ls` from the shell looks like this:

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

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

### 2.9.4 Using ROMFS to Start Applications Automatically

ROMFS can be used with any of the application startup mechanisms simply by referencing the local copy of the application executables. See 2.8.2 Running Applications Automatically, p.66 for information about the various ways in which applications can be run automatically when VxWorks boots.
3.1 Introduction

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources.

Tasks are the basic unit of scheduling in VxWorks. All tasks, whether in the kernel or in processes, are subject to the same scheduler. VxWorks processes are not themselves scheduled.

Intertask communication facilities allow tasks to synchronize and communicate in order to coordinate their activity. In VxWorks, the intertask communication facilities include semaphores, message queues, message channels, pipes, network-transparent sockets, and signals.

For interprocess communication, VxWorks semaphores and message queues, pipes, and events (as well as POSIX semaphores and events) can be created as
public objects to provide accessibility across memory boundaries (between the
kernel and processes, and between different processes). In additions, message
channels provide a socket-based inter-process communications mechanism.

VxWorks provides watchdog timers, but they can only be used in the kernel (see
VxWorks Kernel Programmer’s Guide: Multitasking. However, process-based
applications can use POSIX timers (see 4.8 POSIX Clocks and Timers, p.177).

This chapter discusses the tasking, intertask communication, and interprocess
communication facilities that are at the heart of the VxWorks run-time
environment.

For information about VxWorks and POSIX, see 4. POSIX Facilities.

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

### 3.2 Tasks and Multitasking

VxWorks tasks are the basic unit of code execution in the operating stem itself, as
well as in applications that it executes in processes. In other operating systems the
term thread is used similarly. (For information about VxWorks support for POSIX
threads, see 4.12 POSIX Threads, p.183).

Multitasking provides the fundamental mechanism for an application to control
and react to multiple, discrete real-world events. The VxWorks real-time kernel
provides the basic multitasking environment. Multitasking creates the appearance
of many threads of execution running concurrently when, in fact, the kernel
interleaves their execution on the basis of a scheduling policy.

Each task has its own context, which is the CPU environment and system resources
that the task sees each time it is scheduled to run by the kernel. On a context switch,
a task’s context is saved in the task control block (TCB).

A task’s context includes:

- a thread of execution; that is, the task’s program counter
- the tasks’ virtual memory context (if process support is included)
- the CPU registers and (optionally) coprocessor registers
3.2 Tasks and Multitasking

- stacks for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a time-slice timer
- kernel control structures
- signal handlers
- task variables
- error status (errno)
- debugging and performance monitoring values

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

For more information about virtual memory contexts, see the VxWorks Kernel Programmer’s Guide: Memory Management.

NOTE: The POSIX standard includes the concept of a thread, which is similar to a task, but with some additional features. For details, see 4.12 POSIX Threads, p.183.

3.2.1 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 3-1 describes the state symbols that you see when working with development tools. Example 3-1 shows output from the if() and taskShow() shell commands containing task state information.
## Task State Symbols

<table>
<thead>
<tr>
<th>State Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READY</td>
<td>The task is not waiting for any resource other than the CPU.</td>
</tr>
<tr>
<td>PEND</td>
<td>The task is blocked due to the unavailability of some resource.</td>
</tr>
<tr>
<td>DELAY</td>
<td>The task is asleep for some duration.</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>The task that is unavailable for execution (but not pended or delayed). This state is used primarily for debugging. Suspension does not inhibit state transition, only execution. Thus, pended-suspended tasks can still unblock and delayed-suspended tasks can still awaken.</td>
</tr>
<tr>
<td>STOP</td>
<td>The task is stopped by the debugger.</td>
</tr>
<tr>
<td>DELAY + S</td>
<td>The task is both delayed and suspended.</td>
</tr>
<tr>
<td>PEND + S</td>
<td>The task is both pended and suspended.</td>
</tr>
<tr>
<td>PEND + T</td>
<td>The task is pended with a timeout value.</td>
</tr>
<tr>
<td>STOP + P</td>
<td>Task is pended and stopped (by the debugger, error detection and reporting facilities, or SIGSTOP signal).</td>
</tr>
<tr>
<td>STOP + S</td>
<td>Task is stopped by (by the debugger, error detection and reporting facilities, or SIGSTOP signal) and suspended.</td>
</tr>
<tr>
<td>STOP + T</td>
<td>Task is delayed and stopped (by the debugger, error detection and reporting facilities, or SIGSTOP signal).</td>
</tr>
<tr>
<td>PEND + S + T</td>
<td>The task is pended with a timeout value and suspended.</td>
</tr>
<tr>
<td>STOP+P+S</td>
<td>Task is pended, suspended and stopped by the debugger.</td>
</tr>
<tr>
<td>STOP+P+T</td>
<td>Task pended with a timeout and stopped by the debugger.</td>
</tr>
<tr>
<td>STOP+S+T</td>
<td>Task is suspended with a timeout and stopped by the debugger.</td>
</tr>
<tr>
<td>ST+P+S+T</td>
<td>Task is pended with a timeout, suspended, and stopped by the debugger.</td>
</tr>
<tr>
<td><strong>state + I</strong></td>
<td>The task is specified by <em>state</em> (any state or combination of states listed above), plus an inherited priority.</td>
</tr>
</tbody>
</table>
The STOP state is used by the debugger facilities when a breakpoint is hit. It is also used by the error detection and reporting facilities when an error condition occurs (see 8. Error Detection and Reporting).

Example 3-1  Task States in Shell Command Output

```bash
-> taskShow

            NAME     ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- -------- -------- ------- -----
tShell0    shellTask      455720   1 READY        214118   5db390       0     0
value = 0 = 0x0
-> 1

            NAME     ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- -------- -------- ------- -----
tExcTask   excTask        437460   0 PEND         209fac   484e40       0     0
tJobTask   jobTask        437910   0 PEND         20c6dc   487dd0       0     0
tLogTask   logTask        437c80   0 PEND         209fac   48a190  3d0001     0     0
tShell0    shellTask      455720   1 READY        214118   5db390       0     0
tWdbTask   wdbTask        517a98   3 PEND         20c6dc   5c7560       0     0
tNetTask   netTask        43db90  50 PEND         20c6dc   48d920       0     0
value = 0 = 0x0
->
```
Figure 3-1 illustrates task state transitions for a deployed system—without the STOP state associated with development activity. The routines listed are examples of ones that would cause the associated transition. For example, a task that called taskDelay( ) would move from the ready state to the delayed state.

Figure 3-1  Task State Transitions

The highest-priority ready task is executing.

- ready → pended $\rightarrow$ semTake( ) / msgQReceive( )
- ready → delayed $\rightarrow$ taskDelay( )
- ready → suspended $\rightarrow$ taskSuspend( )
- pended → ready $\rightarrow$ semGive( ) / msgQSend( )
- pended → suspended $\rightarrow$ taskSuspend( )
- delayed → ready $\rightarrow$ expired delay
- delayed → suspended $\rightarrow$ taskSuspend( )
- suspended → ready $\rightarrow$ taskResume( ) / taskActivate( )
- suspended → pended $\rightarrow$ taskResume( )
- suspended → delayed $\rightarrow$ taskResume( )
3.2.2 Task Scheduling

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

- The traditional VxWorks scheduler, which provides priority-based, preemptive scheduling, as well as a round-robin extension. See 3.2.3 VxWorks Traditional Scheduler, p.84.
- The VxWorks POSIX threads scheduler, which is designed (and required) for running pthreads in processes (RTPs). See 4.14 POSIX and VxWorks Scheduling, p.197.
- A custom scheduler framework, which allows you to develop your own scheduler. See the VxWorks Kernel Programmer's Guide: Kernel.

Task Priorities

Task scheduling relies on a task’s priority. The VxWorks kernel provides 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.

A task is assigned its priority at creation, but you can also change it programmatically thereafter. For information about priority assignment, see Task Creation and Activation, p.88 and Task Scheduling Control, p.83).

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

Task Scheduling Control

The routines that control task scheduling are listed in Table 3-2.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskPrioritySet()</td>
<td>Changes the priority of a task.</td>
</tr>
<tr>
<td>taskRtpLock()</td>
<td>Disables task context switching within a process. Prevents any other task in the process from preempting the calling task.</td>
</tr>
<tr>
<td>taskRtpUnLock()</td>
<td>Enables task context switching within a process.</td>
</tr>
</tbody>
</table>
Task Priority

Tasks are assigned a priority when they are created (see Task Creation and Activation, p.88). You can change a task’s priority level while it is executing by calling taskPrioritySet(). The ability to change task priorities dynamically allows applications to track precedence changes in the real world.

Preemption Locks

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

If the task that has disabled the scheduler with taskRtpLock() explicitly blocks or suspends, the scheduler selects the next highest-priority eligible task to execute. When the preemption-locked task unblocks, and begins running again, preemption is again disabled.

If mutual exclusion between tasks in different processes is required, use a public semaphore. For information about global objects, see 3.3.1 Public and Private Objects, p.105.

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 3.3.3 Mutual Exclusion, p.108.

3.2.3 VxWorks Traditional Scheduler

The VxWorks traditional scheduler provides priority-based preemptive scheduling as well as the option of programmatically initiating round-robin scheduling. The traditional scheduler may also be referred to as the original or native scheduler.

The traditional scheduler is included in VxWorks by default with the INCLUDE_VX_TRADITIONAL_SCHEDULER component.
For information about the POSIX thread scheduler and custom schedulers, see 4.14 POSIX and VxWorks Scheduling, p.197 and VxWorks Kernel Programmer’s Guide: Kernel, respectively.

Priority-Based Preemptive Scheduling

A priority-based preemptive scheduler *preempts* the CPU when a task has a higher priority than the current task running. Thus, the kernel ensures that the CPU is always allocated to the highest priority task that is ready to run. This means that if a task—with a higher priority than that of the current task—becomes ready to run, the kernel immediately saves the current task’s context, and switches to the context of the higher priority task. For example, in Figure 3-2, task t1 is preempted by higher-priority task t2, which in turn is preempted by t3. When t3 completes, t2 continues executing. When t2 completes execution, t1 continues executing.

The disadvantage of this scheduling policy is that, when multiple tasks of equal priority must share the processor, if a single task is never blocked, it can usurp the processor. Thus, other equal-priority tasks are never given a chance to run. Round-robin scheduling solves this problem.

![Figure 3-2 Priority Preemption](image)

Round-Robin Scheduling

VxWorks provides a round-robin extension to priority-based preemptive scheduling. Round-robin scheduling accommodates instances in which there are more than one task of a given priority that is ready to run. The round-robin
algorithm attempts to share the CPU amongst these tasks by using time-slicing. Each task in a group of tasks with the same priority executes for a defined interval, or time slice, before relinquishing the CPU to the next task in the group. No one of them, therefore, can usurp the processor until it is blocked. See Figure 3-3 for an illustration of this activity.

Note that while round-robin scheduling is used in some operating systems to provide equal CPU time to all tasks (or processes), regardless of their priority, this is not the case with VxWorks. Priority-based preemption is essentially unaffected by the VxWorks implementation of round-robin scheduling. Any higher-priority task that is ready to run immediately gets the CPU, regardless of whether or not the current task is done with its slice of execution time. When the interrupted task gets to run again, it simply continues using its unfinished execution time.

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

Enabling Round-Robin Scheduling

Round-robin scheduling is enabled by calling kernelTimeSlice(), which takes a parameter for a time slice, or interval.

Time-slice Counts and Preemption

The time-slice or interval defined with a kernelTimeSlice() call is the amount of time that each task is allowed to run before relinquishing the processor to another equal-priority task. Thus, the tasks rotate, each executing for an equal interval of time. No task gets a second slice of time before all other tasks in the priority group have been allowed to run.

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

Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.
If a task blocks or is preempted by a higher priority task during its interval, its time-slice count is saved and then restored when the task becomes eligible for execution. In the case of preemption, the task will resume execution once the higher priority task completes, assuming that no other task of a higher priority is ready to run. In the case where the task blocks, it is placed at the tail of the list of tasks at its priority level. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

Time-slice counts are accrued by the task that is executing when a system tick occurs, regardless of whether or not the task has executed for the entire tick interval. Due to preemption by higher priority tasks or ISRs stealing CPU time from the task, it is possible for a task to effectively execute for either more or less total CPU time than its allotted time slice.
Figure 3-3 shows round-robin scheduling for three tasks of the same priority: \( t_1 \), \( t_2 \), and \( t_3 \). Task \( t_2 \) is preempted by a higher priority task \( t_4 \) but resumes at the count where it left off when \( t_4 \) is finished.

### 3.2.4 Task Creation and Management

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

For interactive use, you can control VxWorks tasks with the host tools or the kernel shell; see the *Wind River Workbench User’s Guide*, the *VxWorks Command-Line Tools User’s Guide*, and *VxWorks Kernel Programmer’s Guide: Target Tools*.

#### Task Creation and Activation

The routines listed in Table 3-3 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:

\[
\text{id} = \text{taskSpawn} ( \text{name}, \text{priority}, \text{options}, \text{stacksize}, \text{main}, \text{arg1}, \ldots, \text{arg10}) ;
\]
Note that a task’s priority can be changed after it has been spawned; see *Task Scheduling Control*, p. 83.

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

The `taskOpen()` routine provides a POSIX-like API for creating a task (with optional activation) or obtaining a handle on existing task. It also provides for creating a task as either a public or private object (see *Task Names and IDs*, p. 91). The `taskOpen()` routine is the most general purpose task-creation routine.

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

### Task Stack

The size of each task’s stack is defined when the task is created (see *Task Creation and Activation*, p. 88).

It can be difficult, however, to know exactly how much stack space to allocate. To help avoid stack overflow and corruption, you can initially allocated a stack that is much larger than you expect the task to require. Then monitor the stack periodically from the shell with `checkStack()` or `ti()`. When you have determined actual usage, adjust the stack size accordingly for testing and for the deployed system.

---

**Table 3-3: Task Creation Routines**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>taskSpawn()</code></td>
<td>Spawns (creates and activates) a new task.</td>
</tr>
<tr>
<td><code>taskCreate()</code></td>
<td>Creates, but not activates a new task.</td>
</tr>
<tr>
<td><code>taskOpen()</code></td>
<td>Open a task (or optionally create one, if it does not exist).</td>
</tr>
<tr>
<td><code>taskActivate()</code></td>
<td>Activates an initialized task.</td>
</tr>
</tbody>
</table>
In addition to experimenting with task stack size, you can also configure and test systems with guard zone protection for task stacks (for more information, see Task Stack Protection, p.90).

Task Stack Protection

Task stacks can be protected with guard zones and by making task stacks non-executable.

Task Stack Guard Zones

Systems can be configured with the INCLUDE_PROTECT_TASK_STACK component to provide guard zone protection for task stacks. If memory usage becomes an issue, the component can be removed for final testing and the deployed system.

An overrun guard zone prevents a task from going beyond the end of its predefined stack size and corrupting data or other stacks. An underrun guard zone typically prevents buffer overflows from corrupting memory that precedes the base of the stack. The CPU generates an exception when a task attempts to access any of the guard zones. The size of a stack is always rounded up to a multiple of the MMU page size when either a guard zone is inserted or when the stack is made non-executable.

Note that guard zones cannot catch instances in which a buffer that causes an overflow is greater than the page size (although this is rare). For example, if the guard zone is one page of 4096 bytes, and the stack is near its end, and then a buffer of a 8000 bytes is allocated on the stack, the overflow will not be detected.

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

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

Note that the INCLUDE_PROTECT_TASK_STACK component does not provide stack protection for tasks that are created with the VX_NO_STACK_PROTECT task
option (see Task Creation Options, p.92). If a task is created with this option, no guard zones are created for that task.

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

- TASK_USER_EXEC_STACK_OVERFLOW_SIZE for user task execution stack overflow size.
- TASK_USER_EXEC_STACK_UNDERFLOW_SIZE for user task execution stack underflow size.
- TASK_USER_EXC_STACK_OVERFLOW_SIZE for user task exception stack overflow size.

The value of these parameters can be modified to increase the size of the guard zones on a system-wide basis. The size of a guard zone is rounded up to a multiple of the CPU MMU page size. The insertion of a guard zone can be prevented by setting the parameter to zero.

Note that for POSIX threads in processes, the size of the execution stack guard zone can be set on an individual basis before the thread is created (using the pthread attributes object, pthread_attr_t). For more information, see 4.12.1 POSIX Thread Stack Guard Zones, p.183.

Task Names and IDs

When a task is spawned, you can specify an ASCII string of any length to be the task name, and a task ID is returned.

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

The following rules and guidelines should be followed when naming tasks:

- The names of public tasks must be unique and must begin with a forward slash; for example /tMyTask. Note that public tasks are visible throughout the entire system—in the kernel and any processes.
- The names of private tasks should be unique. VxWorks does not require that private task names be unique, but it is preferable to use unique names to avoid confusing the user. (Note that private tasks are visible only within the entity in which they were created—either the kernel or a process.)

To use the host development tools to their best advantage, task names should not conflict with globally visible routine or variable names. To avoid name conflicts,
VxWorks uses a convention of prefixing any kernel task name started from the target with the letter t, and any task name started from the host with the letter u. In addition, the name of the initial task of a real-time process is the executable file name (less the extension) prefixed with the letter i.

Creating a task as a public object allows other tasks from outside of its process to send signals or events to it (with the taskKill() or the eventSend() routine, respectively).

For more information, see 3.3.1 Public and Private Objects, p.105.

You do not have to explicitly name tasks. If a NULL pointer is supplied for the name argument of taskSpawn(), then VxWorks assigns a unique name. The name is of the form tN, where N is a decimal integer that is incremented by one for each unnamed task that is spawned.

The taskLib routines listed in Table 3-4 manage task IDs and names.

<table>
<thead>
<tr>
<th>Table 3-4 Task Name and ID Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
</tr>
<tr>
<td>taskName()</td>
</tr>
<tr>
<td>taskNameGet()</td>
</tr>
<tr>
<td>taskNameToId()</td>
</tr>
<tr>
<td>taskIdSelf()</td>
</tr>
<tr>
<td>taskIdVerify()</td>
</tr>
</tbody>
</table>

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

Task Creation Options

When a task is spawned, you can pass in one or more option parameters, which are listed in Table 3-5. The result is determined by performing a logical OR operation on the specified options.
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:

```c
int tid = taskSpawn("tMyTask", 90, VX_FP_TASK, 20000, myFunc, 2387, 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.

### Task Information

The routines listed in Table 3-6 get information about a task by taking a snapshot of a task’s context when the routine is called. Because the task state is dynamic, the information may not be current unless the task is known to be dormant (that is, suspended).
Note that the task-local storage (TLS) facility and the routines provided by tlsLib can be used to maintain information on a task basis.

Also note that each task has a VxWorks events register, which receives events sent from other tasks, ISRs, semaphores, or message queues. See 3.3.7 VxWorks Events, p.126 for more information about this register, and the routines used to interact with it.

Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in Table 3-7 to delete tasks and to protect tasks from unexpected deletion.

Table 3-6 Task Information Routines

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskInfoGet()</td>
<td>Gets information about a task.</td>
</tr>
<tr>
<td>taskPriorityGet()</td>
<td>Examines the priority of a task.</td>
</tr>
<tr>
<td>taskIsSuspended()</td>
<td>Checks whether a task is suspended.</td>
</tr>
<tr>
<td>taskIsReady()</td>
<td>Checks whether a task is ready to run.</td>
</tr>
<tr>
<td>taskIsPended()</td>
<td>Checks whether a task is pended.</td>
</tr>
</tbody>
</table>

Note that the task-local storage (TLS) facility and the routines provided by tlsLib can be used to maintain information on a task basis.

Also note that each task has a VxWorks events register, which receives events sent from other tasks, ISRs, semaphores, or message queues. See 3.3.7 VxWorks Events, p.126 for more information about this register, and the routines used to interact with it.

Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in Table 3-7 to delete tasks and to protect tasks from unexpected deletion.

Table 3-7 Task-Deletion Routines

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit()</td>
<td>Terminates the specified process (and therefore all tasks in it) and frees the process’ memory resources.</td>
</tr>
<tr>
<td>taskExit()</td>
<td>Terminates the calling task (in a process) and frees the stack and any other memory resources, including the task control block.¹</td>
</tr>
<tr>
<td>taskDelete()</td>
<td>Terminates a specified task and frees memory (task stacks and task control blocks only).² The calling task may terminate itself with this routine.</td>
</tr>
<tr>
<td>taskSafe()</td>
<td>Protects the calling task from deletion by any other task in the same process. A task in a different process can still delete that task by terminating the process itself with kill().</td>
</tr>
</tbody>
</table>

¹

²
A process implicitly calls `exit()` function, thus terminating all tasks within it, if the process’ main() routine returns. For more information see 2.5.3 Applications, Processes, and Tasks, p.22.

Tasks implicitly call `taskExit()` if the entry routine specified during task creation returns.

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

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

Using `taskSafe()` to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with `taskSafe()` is blocked. When finished with its critical resource, the protected task can make itself available for deletion by calling `taskUnsafe()`, which readies any deleting task. To support nested deletion-safe regions, a count is kept of the number of times `taskSafe()` and `taskUnsafe()` are called. Deletion is allowed only when the count is zero, that is, there are as many `unsafe`s as `safe`s. 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:

---

**Table 3-7 Task-Deletion Routines (cont’d)**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>taskUnsafe()</code></td>
<td>Undoes a <code>taskSafe()</code>, which makes calling task available for deletion.</td>
</tr>
</tbody>
</table>

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

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

A process implicitly calls `exit()`, thus terminating all tasks within it, if the process’ main() routine returns. For more information see 2.5.3 Applications, Processes, and Tasks, p.22.

Tasks implicitly call `taskExit()` if the entry routine specified during task creation returns.

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

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

Using `taskSafe()` to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with `taskSafe()` is blocked. When finished with its critical resource, the protected task can make itself available for deletion by calling `taskUnsafe()`, which readies any deleting task. To support nested deletion-safe regions, a count is kept of the number of times `taskSafe()` and `taskUnsafe()` are called. Deletion is allowed only when the count is zero, that is, there are as many `unsafe`s as `safe`s. 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:
```c
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.113.

---

### Task Execution Control

The routines listed in Table 3-8 provide direct control over a task’s execution.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskSuspend()</td>
<td>Suspends a task.</td>
</tr>
<tr>
<td>taskResume()</td>
<td>Resumes a task.</td>
</tr>
<tr>
<td>taskRestart()</td>
<td>Restarts a task.</td>
</tr>
<tr>
<td>taskDelay()</td>
<td>Delays a task; delay units are ticks, resolution in ticks.</td>
</tr>
<tr>
<td>nanosleep()</td>
<td>Delays a task; delay units are nanoseconds, resolution in ticks.</td>
</tr>
</tbody>
</table>

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 *4.8 POSIX Clocks and Timers*, p.177.
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.

### 3.2.5 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 3-9; for more information, see the VxWorks API reference for `taskHookLib`.

#### Table 3-9 Task Create, Switch, and Delete Hooks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>taskCreateHookAdd()</code></td>
<td>Adds a routine to be called at every task create.</td>
</tr>
<tr>
<td><code>taskCreateHookDelete()</code></td>
<td>Deletes a previously added task create routine.</td>
</tr>
<tr>
<td><code>taskDeleteHookAdd()</code></td>
<td>Adds a routine to be called at every task delete.</td>
</tr>
<tr>
<td><code>taskDeleteHookDelete()</code></td>
<td>Deletes a previously added task delete routine.</td>
</tr>
</tbody>
</table>

Task create hook routines execute in the context of the creator task.

Task create hooks must consider the ownership of any kernel objects (such as watchdog timers, semaphores, and so on) created in the hook routine. Since create hook routines execute in the context of the creator task, new kernel objects will be
owned by the creator task’s process. It may be necessary to assign the ownership of these objects to the new task’s process. This will prevent unexpected object reclamation from occurring if and when the process of the creator task terminates.

When the creator task is a kernel task, the kernel will own any kernel objects that are created. Thus there is no concern about unexpected object reclamation for this case.

User-installed switch hooks are called within the kernel context and therefore do not have access to all VxWorks facilities. Table 3-10 summarizes the routines that can be called from a task switch hook; in general, any routine that does not involve the kernel can be called.

Table 3-10 Routines Callable by Task Switch Hooks

<table>
<thead>
<tr>
<th>Library</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLib</td>
<td>All routines</td>
</tr>
<tr>
<td>fppArchLib</td>
<td>fppSave(), fppRestore()</td>
</tr>
<tr>
<td>intLib</td>
<td>intContext(), intCount(), intVecSet(), intVecGet(), intLock(), intUnlock()</td>
</tr>
<tr>
<td>lstLib</td>
<td>All routines except lstFree()</td>
</tr>
<tr>
<td>mathALib</td>
<td>All are callable if fppSave()/fppRestore() are used</td>
</tr>
<tr>
<td>rngLib</td>
<td>All routines except rngCreate()</td>
</tr>
<tr>
<td>taskLib</td>
<td>taskIdVerify(), taskIdDefault(), taskIdReady(), taskIdSuspended(), taskTcb()</td>
</tr>
<tr>
<td>vxLib</td>
<td>vxTas()</td>
</tr>
</tbody>
</table>

**NOTE:** For information about POSIX extensions, see 4. POSIX Facilities.

### 3.2.6 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.
A Separate errno Value for Each Task

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

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.

Assignment of Error Status Values

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

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

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

3.2.7 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 (see 8. Error Detection and Reporting).

Tasks can also attach their own handlers for certain hardware exceptions through the signal facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. A user-defined signal handler is useful for recovering from catastrophic events. Typically, setjmp() is called to define the point in the program where control will be restored, and longjmp() is called in the signal handler to restore that context. Note that longjmp() restores the state of the task’s signal mask.

Signals are also used for signaling software exceptions as well as hardware exceptions. They are described in more detail in 3.3.10 Signals, p.149 and in the VxWorks API reference for sigLib.

3.2.8 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 3-4.

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.

The majority of VxWorks routines use the following reentrancy techniques:

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

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

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

**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 3-5.
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. 113.

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 3-6. This facility is provided by the routines taskVarAdd(), taskVarDelete(), taskVarSet(), and taskVarGet(), which are described in the VxWorks API reference for taskVarLib.

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.102 apply to the entire task.

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

In Figure 3-7, 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 3-7  Multiple Tasks Utilizing Same Code**

```c
joint
{
    int jointNum
}
{
    /* joint code here */
}
```
3.3 Intertask and Interprocess Communications

The complement to the multitasking routines described in 3.2 Tasks and Multitasking, p.78 is the intertask communication facilities. These facilities permit independent tasks to coordinate their actions.

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

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

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

3.3.1 Public and Private Objects

Kernel objects such as semaphores and message queues can be created as either private or public objects. This provides control over the scope of their accessibility—which can be limited to a virtual memory context by defining them as private, or extended to the entire system (the kernel and any processes) by defining them as public. There is no difference in performance between a public and a private object.

An object can only be defined as public or private when it is created—the designation cannot be changed thereafter. Public objects must be named when they are created, and the name must begin with a forward slash; for example, /foo. Private objects do not need to be named.

For information about naming tasks in addition to that provided in this section, see Task Names and IDs, p.91.
Creating and Naming Public and Private Objects

Public objects are always named, and the name must begin with a forward-slash. Private objects can be named or unnamed. If they are named, the name must not begin with a forward-slash.

Only one public object of a given class and name can be created. That is, there can be only one public semaphore with the name /foo. But there may be a public semaphore named /foo and a public message queue named /foo. Obviously, more distinctive naming is preferable (such as /fooSem and /fooMQ).

The system allows creation of only one private object of a given class and name in any given memory context; that is, in any given process or in the kernel. For example:

- If process A has created a private semaphore named bar, it cannot create a second semaphore named bar.
- However, process B could create a private semaphore named bar, as long as it did not already own one with that same name.

Note that private tasks are an exception to this rule—duplicate names are permitted for private tasks; see Task Names and IDs, p. 91.

To create a named object, the appropriate xyzOpen() API must be used, such as semOpen(). When the routine specifies a name that starts with a forward slash, the object will be public.

To delete public objects, the xyzDelete() API cannot be used (it can only be used with private objects). Instead, the xyzClose() and xyzUnlink() APIs must be used in accordance with the POSIX standard. That is, they must be unlinked from the name space, and then the last close operation will delete the object (for example, using the semUnlink() and semClose() APIs for a public semaphore). Alternatively, all close operations can be performed first, and then the unlink operation, after which the object is deleted. Note that if an object is created with the OM_DELETE_ON_LAST_CLOSE flag, it is be deleted with the last close operation, regardless of whether or not it was unlinked.

Object Ownership and Resource Reclamation

All objects are owned by the process to which the creator task belongs, or by the kernel if the creator task is a kernel task. When ownership must be changed, for example on a process creation hook, the objOwnerSet() can be used. However, its use is restricted—the new owner must be a process or the kernel.
All objects that are owned by a process are automatically destroyed when the process dies.

All objects that are children of another object are automatically destroyed when the parent object is destroyed.

Processes can share public objects through an object lookup-by-name capability (with the \texttt{xyzOpen()} set of routines). Sharing objects between processes can only be done by name.

When a process terminates, all the private objects that it owns are deleted, regardless of whether or not they are named. All references to public objects in the process are closed (an \texttt{xyzClose()} operation is performed). Therefore, any public object is deleted during resource reclamation, regardless of which process created them, if there are no more outstanding \texttt{xyzOpen()} calls against it (that is, no other process or the kernel has a reference to it), and the object was already unlinked or was created with the \texttt{OM\_DELETE\_ON\_LAST\_CLOSE} option. The exception to this rule is tasks, which are always reclaimed when its creator process dies.

When the creator process of a public object dies, but the object survives because it hasn’t been unlinked or because another process has a reference to it, ownership of the object is assigned to the kernel.

The \texttt{objHandleShow()} show routine can be used to display information about ownership relations between objects in a process.

### 3.3.2 Shared Data Structures

The most obvious way for tasks executing in the same memory space (either a process or the kernel) to communicate is by accessing shared data structures. Because all the tasks in a single process or in the kernel exist in a single linear address space, sharing data structures between tasks is trivial; see Figure 3-8. Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced directly by code running in different contexts.

For information about using shared data regions to communicate between processes, see 2.7 Creating and Managing Shared Data Regions, p.60.
3.3.3 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 4.13 POSIX Thread Mutexes and Condition Variables, p.192.

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:

```c
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
3.4 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 3.3.3 Mutual Exclusion, p.108.
- For *synchronization*, semaphores coordinate a task’s execution with external events.

There are three types of VxWorks 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 semaphores can be created as private objects, which are accessible only within the memory space in which they were created (kernel or process); or as public objects, which accessible throughout the system. For more information, see 3.3.1 Public and Private Objects, p.105.

VxWorks provides not only the semaphores designed expressly for VxWorks, but also POSIX semaphores, designed for portability. An alternate semaphore library provides the POSIX-compliant semaphore interface; see 4.15 POSIX Semaphores, p.208.
The semaphores described here are for use on a single CPU. The optional product VxMP provides semaphores that can be used across processors; see VxWorks Kernel Programmer's Guide: Shared Memory Objects.

Semaphore Control

Instead of defining a full set of semaphore control routines for each type of semaphore, the VxWorks semaphores provide a single uniform interface for semaphore control. Only the creation routines are specific to the semaphore type. Table 3-11 lists the semaphore control routines.

Table 3-11 Semaphore Control Routines

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>semBCreate()</td>
<td>Allocates and initializes a binary semaphore.</td>
</tr>
<tr>
<td>semMCreate()</td>
<td>Allocates and initializes a mutual-exclusion semaphore.</td>
</tr>
<tr>
<td>semCCreate()</td>
<td>Allocates and initializes a counting semaphore.</td>
</tr>
<tr>
<td>semDelete()</td>
<td>Terminates and frees a semaphore.</td>
</tr>
<tr>
<td>semTake()</td>
<td>Takes a semaphore.</td>
</tr>
<tr>
<td>semGive()</td>
<td>Gives a semaphore.</td>
</tr>
<tr>
<td>semFlush()</td>
<td>Unblocks all tasks that are waiting for a semaphore.</td>
</tr>
</tbody>
</table>

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. 113 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 3-9. If the semaphore is available (full), the semaphore becomes unavailable (empty) and the task continues executing immediately. If the semaphore is unavailable (empty), the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

Figure 3-9  Taking a Semaphore

When a task gives a binary semaphore, using semGive(), the outcome also depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 3-10. If the semaphore is already available (full), giving the semaphore has no effect at all. If the semaphore is unavailable (empty) and no task is waiting to take it, then the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).
Mutual Exclusion

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

```c
/* 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:

```c
semTake (semMutex, WAIT_FOREVER);

/* critical region, only accessible by a single task at a time */

semGive (semMutex);
```
Synchronization

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

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

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

Mutual-Exclusion Semaphores

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

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

- It can be used only for mutual exclusion.
- It can be given only by the task that took it.
- The `semFlush()` operation is illegal.

Note that mutex semaphores can be created as user-level objects. These are faster than kernel-level semaphores as long as they are uncontested, which means that:

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

The uncontested case should be the most common given the intended use of a mutex semaphore. In the case when a mutex semaphore is contested, it will be slower than a kernel-level semaphore because a system call must be made.

By default, using the `semMCreate()` routine in a process creates a user-level mutex semaphore. However, a kernel-level semaphore can be created when
semMCreate() is used with the SEM_KERNEL option. The semOpen() routine can only be used in a process to create kernel-level semaphores.

**Priority Inversion**

Figure 3-11 illustrates a situation called priority inversion.

![Figure 3-11 Priority Inversion](image)

**KEY:**
- ▼ = take semaphore
- ◀ = give semaphore
- ◂ = own semaphore
- ▲ = preemption
- ▼ = priority inheritance/release
- ▼ = block

*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 3-11: 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* policy. The priority-inheritance policy assures that a task that holds a resource executes at the priority of the highest-priority task blocked on that resource. Once the task priority has been elevated, it remains at the higher level until all mutual-exclusion semaphores that have contributed to the
tasks elevated priority are released. Hence, the *inheriting* task is protected from preemption by any intermediate-priority tasks. This option must be used in conjunction with a priority queue (SEM_Q_PRIORITY).

In Figure 3-12, priority inheritance solves the problem of priority inversion by elevating the priority of \( t_3 \) to the priority of \( t_1 \) during the time \( t_1 \) is blocked on the semaphore. This protects \( t_3 \), and indirectly \( t_1 \), from preemption by \( t_2 \).

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

```c
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.

```c
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 3-2  **Recursive Use of a Mutual-Exclusion Semaphore**

```c
/* 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 3-12 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

<table>
<thead>
<tr>
<th>Semaphore Call</th>
<th>Count after Call</th>
<th>Resulting Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>semCCreate( )</td>
<td>3</td>
<td>Semaphore initialized with an initial count of 3.</td>
</tr>
<tr>
<td>semTake( )</td>
<td>2</td>
<td>Semaphore taken.</td>
</tr>
<tr>
<td>semTake( )</td>
<td>1</td>
<td>Semaphore taken.</td>
</tr>
<tr>
<td>semTake( )</td>
<td>0</td>
<td>Semaphore taken.</td>
</tr>
<tr>
<td>semTake( )</td>
<td>0</td>
<td>Task blocks waiting for semaphore to be available.</td>
</tr>
<tr>
<td>semGive( )</td>
<td>0</td>
<td>Task waiting is given semaphore.</td>
</tr>
<tr>
<td>semGive( )</td>
<td>1</td>
<td>No task waiting for semaphore; count incremented.</td>
</tr>
</tbody>
</table>

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 VxWorks semaphore interface includes three special options. These options are not available for the POSIX-compliant semaphores described in 4.15 POSIX Semaphores, p.208.

**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**

VxWorks semaphores include the ability to select the queuing mechanism employed for tasks blocked on a semaphore. They can be queued based on either of two criteria: first-in first-out (FIFO) order, or priority order; see **Figure 3-13**.

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

By default, a task that receives a signal while pending on a semaphore, executes the associated signal handler, and then returns to pending on the semaphore.

The SEM_INTERRUPTIBLE option for counting binary and mutex semaphores changes this behavior. When a task receives a signal while pending on a semaphore that was created with the SEM_INTERRUPTIBLE option, the associated signal handler is executed, as with the default behavior. However, the semTake() call then returns ERROR with errno set to EINTR to indicate to the caller than a signal occurred while pending on the semaphore.

Semaphores and VxWorks Events

Semaphores can send VxWorks events to a specified task when they becomes free. For more information, see 3.3.7 VxWorks Events, p.126.

3.3.5 Message Queues

Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In VxWorks, the primary intertask communication mechanism within a single CPU is message queues.

For information about socket-based message communication across memory spaces (kernel and processes), and between multiple nodes, see 3.3.8 Message Channels, p.133.

Message queues allow a variable number of messages, each of variable length, to be queued. Tasks and ISRs can send messages to a message queue, and tasks can receive messages from a message queue.
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 3-14.

VxWorks message queues can be created as private objects, which accessible only within the memory space in which they were created (process or kernel); or as public objects, which accessible throughout the system. For more information, see 3.3.1 Public and Private Objects, p.105.

There are two message-queue subroutine libraries in VxWorks. The first of these, msgQLib, provides VxWorks message queues, designed expressly for VxWorks; the second, mqPxLib, is compliant with the POSIX standard (1003.1b) for real-time extensions. See 4.15.1 Comparison of POSIX and VxWorks Semaphores, p.209 for a discussion of the differences between the two message-queue designs.

**VxWorks Message Queues**

VxWorks message queues are created, used, and deleted with the routines shown in Table 3-13. 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.

---

*Figure 3-14  Full Duplex Communication Using Message Queues*

```
message queue 1

message

task 1

message

message queue 2

message

task 2
```
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 3-3  **VxWorks Message Queues**

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

/* includes */
#include <vxWorks.h>
#include <msgQLib.h>

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

MSG_Q_ID myMsgQId;

/* 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:
%s
", msgBuf);

#define MESSAGE "Greetings from Task 1"

/* 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);

**Interruptible**

By default, a task that receives a signal while pending on a message queue, executes the associated signal handler, and then returns to pending.

The **MSG_Q_INTERRUPTIBLE** option for message queues changes this behavior. When a task receives a signal while pending on a message queue that was created with the **MSG_Q_INTERRUPTIBLE** option, the associated signal handler is executed, as with the default behavior. However, the msgQSend() or
msgQReceive( ) call then returns ERROR with errno set to EINTR to indicate to the caller than a signal occurred while pending on the message queue.

Queuing

VxWorks message queues include the ability to select the queuing mechanism employed for tasks blocked on a message queue. The MSG_Q_FIFO and MSG_Q_PRIORITY options are provided to specify (to the msgQCreate() and msgQOpen() routines) the queuing mechanism that should be used for tasks that pend on msgQSend() and msgQReceive().

Displaying Message Queue Attributes

The VxWorks show() command produces a display of the key message queue attributes, for either kind of message queue. For example, if myMsgQId is a VxWorks message queue, the output is sent to the standard output device, and looks like the following from the shell (using the C interpreter):

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

Servers and Clients with Message Queues

Real-time systems are often structured using a client-server model of tasks. In this model, server tasks accept requests from client tasks to perform some service, and usually return a reply. The requests and replies are usually made in the form of intertask messages. In VxWorks, message queues or pipes (see 3.3.6 Pipes, p.125) are a natural way to implement this functionality.

For example, client-server communications might be implemented as shown in Figure 3-15. 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.

Message Queues and VxWorks Events

Message queues can send VxWorks events to a specified task when a message arrives on the queue and no task is waiting on it. For more information, see 3.3.7 VxWorks Events, p.126.

3.3.6 Pipes

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

The created pipe is a normally named I/O device. Tasks can use the standard I/O routines to open, read, and write pipes, and invoke ioctl routines. As they do with other I/O devices, tasks block when they read from an empty pipe until data is available, and block when they write to a full pipe until there is space available.

As I/O devices, pipes provide one important feature that message queues cannot—the ability to be used with select(). This routine allows a task to wait for data to be available on any of a set of I/O devices. The select() routine also works with other asynchronous I/O devices including network sockets and serial devices. Thus, by using select(), a task can wait for data on a combination of several pipes, sockets, and serial devices; see 6.3.9 Pending on Multiple File Descriptors: The Select Facility, p.272.

Pipes allow you to implement a client-server model of intertask communications; see Servers and Clients with Message Queues, p.124.

3.3.7 VxWorks Events

VxWorks events provide a means of communication and synchronization between tasks and other tasks, interrupt service routines (ISRs) and tasks, semaphores and tasks, and message queues and tasks.¹

Events can be used as a lighter-weight alternative to binary semaphores for task-to-task and ISR-to-task synchronization (because no object must be created). They can also be used to notify a task that a semaphore has become available, or that a message has arrived on a message queue.

The events facility provides a mechanism for coordinating the activity of a task using up to thirty-two events that can be sent to it by other tasks, ISRs, semaphores, and message queues. A task can wait on multiple events from multiple sources. Events thereby provide a means for coordination of complex matrix of activity without allocation of additional system resources.

Each task has 32 event flags, bit-wise encoded in a 32-bit word (bits 25 to 32 are reserved for Wind River use). These flags are stored in the task’s event register. Note that an event flag itself has no intrinsic meaning. The significance of each of the 32 event flags depends entirely on how any given task is coded to respond to their being set. There is no mechanism for recording how many times any given event

¹. VxWorks events are based on pSOS operating system events. VxWorks introduced functionality similar to pSOS events (but with enhancements) with the VxWorks 5.5 release.
has been received by a task. Once a flag has been set, its being set again by the same or a different sender is essentially an invisible operation.

Events are similar to signals in that they are sent to a task asynchronously; but differ in that receipt is synchronous. That is, the receiving task must call a routine to receive at will, and can choose to pend while waiting for events to arrive. Unlike signals, therefore, events do not require a handler.

For a code example of how events can be used, see the eventLib API reference.

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

### Configuring VxWorks for Events

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

### Preparing a Task to Receive Events

A task can pend on one or more events, or simply check on which events have been received, with a call to `eventReceive()` . The routine specifies which events to wait for, and provides options for waiting for one or all of those events. It also provides various options for how to manage unsolicited events.

In order for a task to receive events from a semaphore or a message queue, however, it must first register with the specific object, using `semEvStart()` for a semaphore or `msgQEvStart()` for a message queue. Only one task can be registered with any given semaphore or message queue at a time.

The `semEvStart()` routine identifies the semaphore and the events that it should send to the task when the semaphore is free. It also provides a set of options to specify whether the events are sent only the first time the semaphore is free, or each time; whether to send events if the semaphore is free at the time of registration; and whether a subsequent `semEvStart()` call from another task is allowed to take effect (and to unregister the previously registered task).

Once a task has registered with a semaphore, every time the semaphore is released with `semGive()` , and as long as no other tasks are pending on it, the semaphore sends events to the registered task.

To request that the semaphore stop sending events to it, the registered task calls `semEvStop()` .
Registration with a message queue is similar to registration with a semaphore. The `msgQEvStart()` routine identifies the message queue and the events that it should send to the task when a message arrives and no tasks are pending on it. It provides a set of options to specify whether the events are sent only the first time a message is available, or each time; whether a subsequent call to `msgQEvStart()` from another task is allowed to take effect (and to unregister the previously registered task).

Once a task has registered with a message queue, every time the message queue receives a message and there are no tasks pending on it, the message queue sends events to the registered task.

To request that the message queue stop sending events to it, the registered task calls `msgQEvStop()`.

**Sending Events to a Task**

Tasks and ISRs can send specific events to a task using `eventSend()`, whether or not the receiving task is prepared to make use of them.

Semaphores and message queues send events automatically to tasks that have registered for notification with `semEvStart()` or `msgQEvStart()`, respectively. These objects send events when they are free. The conditions under which objects are free are as follows:

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

**Binary Semaphore**
A binary semaphore is considered free when no task owns it and no task is waiting for it.

**Counting Semaphore**
A counting semaphore is considered free when its count is nonzero and no task is pending on it. Events cannot, therefore, be used as a mechanism to compute the number of times a semaphore is released or given.

**Message Queue**
A message queue is considered free when a message is present in the queue and no task is pending for the arrival of a message in that queue. Events
cannot, therefore, be used as a mechanism to compute the number of messages sent to a message queue.

Note that just because an object has been released does not mean that it is free. For example, if a semaphore is given, it is released; but it is not free if another task is waiting for it at the time it is released. When two or more tasks are constantly exchanging ownership of an object, it is therefore possible that the object never becomes free, and never sends events.

Also note that when a semaphore or message queue sends events to a task to indicate that it is free, it does not mean that the object is in any way reserved for the task. A task waiting for events from an object unpends when the resource becomes free, but the object may be taken in the interval between notification and unpending. The object could be taken by a higher priority task if the task receiving the event was pended in `eventReceive()`. Or a lower priority task might steal the object: if the task receiving the event was pended in some routine other than `eventReceive()`, a low priority task could execute and (for example) perform a `semTake()` after the event is sent, but before the receiving task unpends from the blocking call. There is, therefore, no guarantee that the resource will still be available when the task subsequently attempts to take ownership of it.

**WARNING:** Because events cannot be reserved for an application in any way, care should be taken to ensure that events are used uniquely and unambiguously. Note that events 25 to 32 (VXE25 to VXEV32) are reserved for Wind River’s use, and should not be used by customers. Third parties should be sure to document their use of events so that their customers do not use the same ones for their applications.

**Events and Object Deletion**

If a semaphore or message queue is deleted while a task is waiting for events from it, the task is automatically unpended by the `semDelete()` or `msgQDelete()` implementation. This prevents the task from pending indefinitely while waiting for events from an object that has been deleted. The pending task then returns to the ready state (just as if it were pending on the semaphore itself) and receives an ERROR return value from the `eventReceive()` call that caused it to pend initially.

If, however, the object is deleted between a tasks’ registration call and its `eventReceive()` call, the task pends anyway. For example, if a semaphore is deleted while the task is between the `semEvStart()` and `eventReceive()` calls, the task pends in `eventReceive()`, but the event is never sent. It is important, therefore, to use a timeout other than `WAIT_FOREVER` when object deletion is expected.
Events and Task Deletion

If a task is deleted before a semaphore or message queue sends events to it, the events can still be sent, but are obviously not received. By default, VxWorks handles this event-delivery failure silently.

It can, however, be useful for an application that created an object to be informed when events were not received by the (now absent) task that registered for them. In this case, semaphores and message queues can be created with an option that causes an error to be returned if event delivery fails (the SEM_EVENTSEND_ERROR_NOTIFY and MSG_Q_EVENTSEND_ERROR_NOTIFY options, respectively). The semGive() or msgQSend() call then returns ERROR when the object becomes free.

The error does not mean the semaphore was not given or that the message was not properly delivered. It simply means the resource could not send events to the registered task. Note that a failure to send a message or give a semaphore takes precedence over an events failure.

Accessing Event Flags

When events are sent to a task, they are stored in the task’s events register (see Task Events Register, p.131), which is not directly accessible to the task itself.

When the events specified with an eventReceive() call have been received and the task unpends, the contents of the events register is copied to a variable that is accessible to the task.

When eventReceive() is used with the EVENTS_WAIT_ANY option—which means that the task unpends for the first of any of the specified events that it receives—the contents of the events variable can be checked to determine which event caused the task to unpend.

The eventReceive() routine also provides an option that allows for checking which events have been received prior to the full set being received.
Events Routines

The routines used for working with events are listed in Table 3-14.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventSend( )</td>
<td>Sends specified events to a task.</td>
</tr>
<tr>
<td>eventReceive( )</td>
<td>Pends a task until the specified events have been received. Can also be used to check what events have been received in the interim.</td>
</tr>
<tr>
<td>eventClear( )</td>
<td>Clears the calling task’s event register.</td>
</tr>
<tr>
<td>semEvStart( )</td>
<td>Registers a task to be notified of semaphore availability.</td>
</tr>
<tr>
<td>semEvStop( )</td>
<td>Unregisters a task that had previously registered for notification of semaphore availability.</td>
</tr>
<tr>
<td>msgQEvStart( )</td>
<td>Registers a task to be notified of message arrival on a message queue when no recipients are pending.</td>
</tr>
<tr>
<td>msgQEvStop( )</td>
<td>Unregisters a task that had previously registered for notification of message arrival on a message queue.</td>
</tr>
</tbody>
</table>

For more information about these routines, see the VxWorks API references for eventLib, semEvLib, and msgQEvLib.

Task Events Register

Each task has its own task events register. The task events register is a 32-bit field used to store the events that the task receives from other tasks (or itself), ISRs, semaphores, and message queues.

Events 25 to 32 (VXEV25 or 0x01000000 to VXEV32 or 0x80000000) are reserved for Wind River use only, and should not be used by customers.

As noted above (Accessing Event Flags, p.130), a task cannot access the contents of its events registry directly.

Table 3-15 describes the routines that affect the contents of the events register.
Show Routines and Events

For the purpose of debugging systems that make use of events, the `taskShow`, `semShow`, and `msgQShow` libraries display event information.

The `taskShow` library displays the following information:

- the contents of the event register
- the desired events
- the options specified when `eventReceive()` was called

The `semShow` and `msgQShow` libraries display the following information:

- the task registered to receive events
- the events the resource is meant to send to that task
- the options passed to `semEvStart()` or `msgQEvStart()`

<table>
<thead>
<tr>
<th>Routine</th>
<th>Effect on the Task Events Register</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventReceive()</code></td>
<td>Clears or leaves the contents of the task’s events register intact, depending on the options selected.</td>
</tr>
<tr>
<td><code>eventClear()</code></td>
<td>Clears the contents of the task’s events register.</td>
</tr>
<tr>
<td><code>eventSend()</code></td>
<td>Writes events to a task’s events register.</td>
</tr>
<tr>
<td><code>semGive()</code></td>
<td>Writes events to the task’s events register, if the task is registered with the semaphore.</td>
</tr>
<tr>
<td><code>msgQSend()</code></td>
<td>Writes events to a task’s events register, if the task is registered with the message queue.</td>
</tr>
</tbody>
</table>
3.3.8 Message Channels

Message channels are a socket-based facility that provides for inter-process communication across memory boundaries on a single node (processor), as well as for inter-process communication between multiple nodes (multi-processor). That is, message channel communications can take place between tasks running in the kernel and tasks running in processes (RTPs) on a single node, as well as between multiple nodes, regardless of the memory context in which the tasks are running. For example, message channels can be used to communicate between:

- a task in the kernel and a task in a process on a single node
- a task in one process and a task in another process on a single node
- a task in the kernel of one node and a task in a process on another node
- a task in a process on one node and a task in a process on another node

and so on.

The scope of message channel communication can be configured to limit server access to:

- one memory space on a node (either the kernel or one process)
- all memory spaces on a node (the kernel and all processes)
- a cluster of nodes in a system (including all memory spaces in each node)

Message channels provide a connection-oriented messaging mechanism. Tasks exchange information in the form of messages that can be of variable size and format. They can be passed back and forth in full duplex mode once the connection is established. Message channels can also provide a connection-oriented messaging mechanism between separate nodes in a cluster.

Message Channel Facilities

The message channel technology consists of the following basic facilities:

- The Transparent Inter-Process Communication (TIPC) infrastructure for multi-node communication. See Multi-Node Communication with TIPC, p.137.
- The Socket Name Service (SNS), which provides location and interface transparency for message channel communication between tasks on a single node, and maintains communication between nodes for multi-node message channel communications. In addition, it controls the scope of message channel
communication (to two memory spaces, a node, or a cluster of nodes). See *Socket Name Service*, p.138.

- The Socket Application Libraries (SAL), which provide APIs for using message channels in applications, as well as the mechanism for registering the tasks that are using a message channel with a Socket Name Service. See *Socket Application Libraries*, p.142.

Also see *Comparing Message Channels and Message Queues*, p.148.

**Single-Node Communication with COMP**

The underlying transport mechanism for single-node message channels is based on the Connection-Oriented Message Passing protocol (COMP), which provides a fast method for transferring messages across memory boundaries on a single node.

COMP is designed for use with the standard socket API. Because it provides connection-oriented messaging, the socket type associated with message channels is the `SOCK_SEQPACKET`. The protocol is connection-based, like other stream-based protocols such as TCP, but it carries variable-sized messages, like datagram-based protocols such as UDP.

While COMP provides for standard socket support, it has no dependency on TCP/IP networking facilities, which can be left out of a system if the facilities are not otherwise needed.

In providing single-node local communications, COMP sockets are available as part of the `AF_LOCAL` domain. Although this domain is traditionally related to the UNIX file system, in VxWorks the addressing is completely independent of any file system. Like UNIX sockets, COMP uses a string to define the address, and it has a structure similar to a file path name, but this is the extent of the similarity in this regard. The address is simply a logical representation of the end-point.

The transfer of data in message channels is based on an internal buffer management implementation that allows for deterministic memory allocation, which reduces the amount of copies needed to transfer the data whenever possible. Only one copy is needed for the internal transfer; the data coming from the user is directly moved into the receiver buffer space. Another copy is required to submit and retrieve the data to and from the channel.

COMP supports the standard socket options, such as `SO_SNDBUF` or `SO_RECVBUF` and `SO_RCVTIMEO` and `SO_SNDTIMEO`. For information about the socket options, see the socket API references. For information about how COMP uses them, see `installDir/vxworks-6.x/target/src/dsi/backend/dsiSockLib.c`. 
Express Messaging

Express messaging is also available for sending and receiving a message. An express message is placed on a special queue on the sending side and placed at the front of the normal queue at the receiving end. This allows for urgent messages to be sent and received with a higher priority than the normal messages. In order to send an express message, the `flags` parameter of the standard `send()` routine must have the `MSG_EXP` bit set. (Also see the `socket send()` API reference).

Show Routines

Because COMP is based on the standard socket API, traditional network show routines can be used, such as `netstat()`. In addition, information on local sockets can be retrieved with the `unstatShow()` routine.

COMP Socket Support with DSI

The COMP socket functional interface is provided by the DSI back end. The back end provides the set of implementations of the standard socket functions for the COMP protocol specific calls. The traditional network protocols in VxWorks, such as TCP and UDP, use the BSD Internet Domain Socket back end and are described in the *Wind River Network Stack for VxWorks 6 Programmer’s Guide*. The DSI back end is a simplified version of the BSD back-end. It is designed for optimized communications when both end points are in a single node (which is true for COMP).

The DSI back end requires its own system and data memory pools, which are used to handle the creation of sockets and the data transfers between two endpoints. The pools are similar to those required for the network stack. In addition, the pools are configured so as to enhance performance for the local transfers. The system pool provides COMP with the memory it needs for its internal structures and data types. The data pool provides COMP with the memory it needs for receiving data. Because COMP is local, data transfer has been optimized so that data are put directly in the receiver’s packet queue.

Both the DSI back end and DSI memory pools complement the BSD equivalent. Therefore, both BSD and DSI sockets can coexist in the system. They do not depend on each other, so that they can be added or removed, as needed.

COMP uses `netBufLib` to manage its internal system and data memory pools. For detailed information on how buffers are configured, see the coverage of the similar technology, `netBufPool`, in the *Wind River Network Stack for VxWorks 6 Programmer’s Guide*. 
These pools are created automatically by the `INCLUDE_DSI_POOL` component. The DSI parameters listed in Table 3-16 are used for memory pool configuration. These parameters are used when `usrNetDsiPoolConfig()` routine is called, which happens automatically when the system boots. The `dsiSysPoolShow()` and `dsiDataPoolShow()` can be used to display related information (see the VxWorks API reference for `dsiSockLib`).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSI_NUM_SOCKETS</td>
<td>200</td>
</tr>
<tr>
<td>DSI_DATA_32</td>
<td>50</td>
</tr>
<tr>
<td>DSI_DATA_64</td>
<td>100</td>
</tr>
<tr>
<td>DSI_DATA_128</td>
<td>200</td>
</tr>
<tr>
<td>DSI_DATA_256</td>
<td>40</td>
</tr>
<tr>
<td>DSI_DATA_512</td>
<td>40</td>
</tr>
<tr>
<td>DSI_DATA_1K</td>
<td>10</td>
</tr>
<tr>
<td>DSI_DATA_2K</td>
<td>10</td>
</tr>
<tr>
<td>DSI_DATA_4K</td>
<td>10</td>
</tr>
<tr>
<td>DSI_DATA_8K</td>
<td>10</td>
</tr>
<tr>
<td>DSI_DATA_16K</td>
<td>4</td>
</tr>
<tr>
<td>DSI_DATA_32K</td>
<td>0</td>
</tr>
<tr>
<td>DSI_DATA_64K</td>
<td>0</td>
</tr>
</tbody>
</table>

The DSI pool is configured more strictly and more efficiently than the core network pool since it is more contained, fewer scenarios are possible, and everything is known in advance (as there is only the one node involved). The `DSI_NUM_SOCKETS` parameter controls the size of the system pool. It controls the number of clusters needed to fit a socket, for each family and each protocol supported by the back end. Currently, only the `AF_LOCAL` address family is supported by COMP.

The clusters allocated in the back end are of these sizes:
3 Multitasking
3.3 Intertask and Interprocess Communications

- aligned sizeof (struct socket)
- aligned sizeof (struct uncompcb)
- aligned sizeof (struct sockaddr_un)

One cluster of size 328 and of size 36 are needed for each socket that is created since currently, the COMP protocol is always linked to a DSI socket. Only one cluster of sizeof (struct sockaddr_un) is required, therefore the size of the system pool is basically determined by: (DSI_NUM_SOCKETS * (328 + 36) + 108)). Using these sizes prevents any loss of space since they are the actual sizes needed.

All other parameters for the DSI pool are used to calculate the size of clusters in the data pool, and at the same time, the size of the pool itself. The data pool is used as packet holders during the transmissions between two sockets, between the time the data is copied from the sender’s buffer to the receiver’s buffer. Each of them represent a cluster size from 32 bytes to 64 kilobytes and the number of allocated clusters of that specific size.

To set reasonable values for the parameters in this component, you must know how much memory your deployed application will require. There is no simple formula that you can use to anticipate memory usage. Your only real option is to determine memory usage empirically. This means running your application under control of the debugger, pausing the application at critical points in its execution, and monitoring the state of the memory pool. You will need to perform these tests under both stressed and unstressed conditions.

Multi-Node Communication with TIPC

The underlying transport mechanism for multi-node message channels is based on the Transparent Inter-Process Communication (TIPC) protocol, which provides a fast method for transferring messages across node boundaries in a cluster environment. TIPC can also be used within a single node.

TIPC is designed for use with the standard socket API. For connection-oriented messaging, the socket type associated with message channels is the SOCK_SEQPACKET.

The TIPC protocol is connection-based, like other stream-based protocols such as TCP, but it carries variable-sized messages, like datagram-based protocols such as UDP. In providing cluster and node based communications, TIPC sockets are available in the AF_TIPC domain. TIPC provides several means of identifying endpoints that are handled transparently through the SNS name server. In this release, TIPC has a dependency on the TCP/IP stack. For more information about TIPC, see the Wind River TIPC for VxWorks 6 Programmer’s Guide.
TIPC Socket Support With BSD

The TIPC socket functionality is based on the BSD socket back end.

Socket Name Service

A Socket Name Service (SNS) allows a server application to associate a service name with a collection of listening sockets, as well as to limit the visibility of the service name to a restricted (but not arbitrary) set of clients.

Both Socket Application Library (SAL) client and server routines make use of an SNS server to establish a connection to a specified service without the client having to be aware of the address of the server’s listening sockets, or the exact interface type being utilized (see Socket Application Libraries, p. 142). This provides both location transparency and interface transparency. Such transparency makes it possible to design client and server applications that can operate efficiently without requiring any knowledge of the system’s topology.

An SNS server is a simple database that provides an easy mapping of service names and their associated sockets. The service name has this URL format:

```
[SNS:] service_name [ @scope ]
```

The [SNS:] prefix is the only prefix accepted, and it can be omitted. The scope can have the following values: private, node, or cluster. These values designate an access scope for limiting access to the same single memory space (the kernel or a process), the same node (the kernel and all processes on that node), and a set of nodes, respectively. A server can be accessed by clients within the scope that is defined when the server is created with the salCreate() routine (see SAL Server Library, p. 143).

SNS provides a resource reclamation mechanism for servers created within processes. If a process dies before salDelete() has been called on a SAL server, SNS will be notified and will remove the entry from the database. Note, however, that this mechanism is not available for tasks in the kernel. If a task in the kernel terminates before salDelete() is called, the service name is not automatically removed from SNS. In order to avoid stale entries that may prevent new services with the same name from being created, the salRemove() routine should be used.

The SNS server can be configured to run in either kernel or user space. A node should not be configured with more than one SNS server. The server starts at boot time, and is named tSnsServer if it is running in the kernel, or iSnsServer if it is running as a process. For a multi-node system, a monitoring task is automatically
spawned to maintain a list of all the SNS servers in the cluster. The monitoring task is named \texttt{tDsalMonitor}, and it runs in the kernel.

The \texttt{snsShow()} command allows a user to verify that SAL-based services are correctly registered with the SNS server from the shell (see \texttt{snsShow()} Example, p.140).

For more information, see the VxWorks API reference for \texttt{snsLib}.

**Multi-Node Socket Name Service**

For a multi-node system, a Socket Name Service (SNS) runs on each node that is configured to use SAL. Note that VxWorks SNS components for multi-node use are different from those used on single node systems (see Configuring VxWorks for Message Channels, p.145).

When a distributed SNS server starts on a node at boot time, it uses a TIPC bind operation to publish a TIPC port name. This is visible to all other nodes in the cluster. The other existing SNS servers then register the node in their tables of SNS servers. A separate monitoring task (called \texttt{tDsalMonitor}) is started on each node at boot time, which uses the TIPC subscription feature to detect topology-change events such as a new SNS server coming online, or an existing SNS server leaving the cluster.

Note that if the TIPC networking layer does not start up properly at boot time, the distributed SAL system will not initialize itself correctly with TIPC, and the SNS server will work strictly in local mode. The SNS server does not check for a working TIPC layer after the system boots, so that it will not detect the layer if it is subsequently started manually, and the SNS server will continue to run in local mode.

When a new node appears, each SNS server sends a command to that node requesting a full listing of all sockets that are remotely accessible. The SNS server on the new node sends a list of sockets that can be reached remotely.

Each time a new socket is created with \texttt{salCreate()} on a node that has a server scope greater than \texttt{node}, this information is sent to all known SNS servers in the cluster. All SNS servers are thereby kept up to date with relevant information. Similarly, when a socket is deleted using the \texttt{salRemove()} function, this information is sent to all known SNS servers in the cluster. The addition and removal of sockets is an infrequent occurrence in most anticipated uses and should be of minimal impact on network traffic and on the performance of the node.

When the \texttt{tDsalMonitor} task detects that an SNS server has been withdrawn from the system, the local SNS server purges all entries related to the node that is no longer a part of the distributed SNS cluster.
Note that only information on accessible sockets is transmitted to remote SNS servers. While it is acceptable to create an AF_LOCAL socket with cluster scope, this socket will use the COMP protocol which can only be accessed locally. SNS servers on remote nodes will not be informed of the existence of this socket.

On a local node, if a socket name exists in the SNS database in both the AF_LOCAL and AF_TIPC families, when a connection is made to that name using salOpen(), the AF_LOCAL socket will be used.

**snsShow( ) Example**

The snsShow() shell command provides information about all sockets that are accessible from the local node, whether the sockets are local or remote. The command is provided by the VxWorks INCLUDE_SNS_SHOW component.
The following examples illustrate `snsShow()` output from three different nodes in a system.

### From Node <1.1.22>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SCOPE</th>
<th>FAMILY</th>
<th>TYPE</th>
<th>PROTO</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>astronaut_display</td>
<td>clust</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x5</code></td>
</tr>
<tr>
<td>ground_control_timestamp</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717967</code></td>
</tr>
<tr>
<td>ground_control_weblog</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.25&gt;,1086717965</code></td>
</tr>
<tr>
<td>heartbeat_private</td>
<td>priv</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x4</code></td>
</tr>
<tr>
<td>local_temperature</td>
<td>node</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x2</code></td>
</tr>
<tr>
<td>newsfeed</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.25&gt;,1086717965</code></td>
</tr>
<tr>
<td>rocket_diagnostic_port</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717964</code></td>
</tr>
<tr>
<td>rocket_propellant_fuel_level_interface</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717960</code></td>
</tr>
<tr>
<td>spacestation_docking_port</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717960</code></td>
</tr>
</tbody>
</table>

### From Node <1.1.25>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SCOPE</th>
<th>FAMILY</th>
<th>TYPE</th>
<th>PROTO</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>astronaut_display</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717967</code></td>
</tr>
<tr>
<td>ground_control_timestamp</td>
<td>clust</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x3</code></td>
</tr>
<tr>
<td>ground_control_weblog</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.25&gt;,1086717965</code></td>
</tr>
<tr>
<td>local_billboard</td>
<td>node</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x2</code></td>
</tr>
<tr>
<td>newsfeed</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.50&gt;,1086717962</code></td>
</tr>
<tr>
<td>rocket_diagnostic_port</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717964</code></td>
</tr>
<tr>
<td>rocket_propellant_fuel_level_interface</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717960</code></td>
</tr>
<tr>
<td>spacestation_docking_port</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717960</code></td>
</tr>
</tbody>
</table>

### From Node <1.1.55>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SCOPE</th>
<th>FAMILY</th>
<th>TYPE</th>
<th>PROTO</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>astronaut_display</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717967</code></td>
</tr>
<tr>
<td>ground_control_timestamp</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.25&gt;,1086717965</code></td>
</tr>
<tr>
<td>ground_control_weblog</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.25&gt;,1086717961</code></td>
</tr>
<tr>
<td>newsfeed</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.50&gt;,1086717962</code></td>
</tr>
<tr>
<td>rocket_diagnostic_port</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717964</code></td>
</tr>
<tr>
<td>rocket_propellant_fuel_level_interface</td>
<td>clust</td>
<td>TIPC</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>&lt;1.1.22&gt;,1086717960</code></td>
</tr>
<tr>
<td>spacestation_docking_port</td>
<td>clust</td>
<td>LOCAL</td>
<td>SEQPKT</td>
<td>0</td>
<td><code>/comp/socket/0x2</code></td>
</tr>
</tbody>
</table>

The output of the `snsShow()` command is fairly self-explanatory. The first field is the name of the socket. If the name is longer than the space allocated in the output,
the entire name is printed and the other information is presented on the next line with the name field containing several dashes.

The scope values are `priv` for private, `node` for node, and `clust` for cluster.

The family types can be `TIPC` for `AF_TIPC` or `LOCAL` for `AF_LOCAL`.

The socket type can be `SEQPKT` for `SOCK_SEQPACKET`, `RDM`.

The protocol field displays a numeric value and a location indicator. The numeric value is reserved for future use, and currently only zero is displayed. The final character in the field indicates whether the socket was created on a remote or local node, with an asterisk (*) designating remote.

The address field indicates the address of the socket. All addresses of the form `/comp/socket` belong to the `AF_LOCAL` family. All addresses of the form `<x.y.z>,refID` belong to the `AF_TIPC` family. The TIPC address gives the TIPC `portID` which consists of the `nodeID` and the unique reference number.

**Socket Application Libraries**

The Socket Application Libraries (SAL) simplify creation of both server and client applications by providing routines to facilitate use of the sockets API.

SAL also provides an infrastructure for the development of location-transparent and interface-transparent applications. By allowing SAL to handle the basic housekeeping associated with a socket-based application, developers can focus on the application-specific portions of their designs. Developers are free to use the complete range of SAL capabilities in their applications, or just the subset that suits their needs; they can even bypass SAL entirely and develop a socket-based application using nothing but custom software. The SAL client and server APIs can be used in both kernel and user space.

Several VxWorks components are available to provide SAL support in different memory spaces, for single or multi-node systems, and so on (see *Configuring VxWorks for Message Channels*, p.145).

SAL-based applications can also utilize the Socket Name Service (SNS), which allows a client application to establish communication with a server application without having to know the socket addresses used by the server (see *Socket Name Service*, p.138).
SAL Server Library

The SAL server routines provide the infrastructure for implementing a socket-based server application. The SAL server allows a server application to provide service to any number of client applications. A server application normally utilizes a single SAL server in its main task, but is free to spawn additional tasks to handle the processing for individual clients if parallel processing of client requests is required. The SAL server library is made of the following routines:

- **salCreate()**
  Creates a named socket-based server.

- **salDelete()**
  Deletes a named socket-based server.

- **salServerRtnSet()**
  Configures the processing routine with the SAL server.

- **salRun()**
  Activates a socket-based server.

- **salRemove()**
  Removes a service from the SNS by name.

A server application typically calls **salCreate()** to configure a SAL server with one or more sockets that are then automatically registered with SNS under a specified service identifier. The number of sockets created depends on which address families, socket types, and socket protocols are specified by the server application. **AF_LOCAL** and **AF_TIPC** sockets are supported.

If the address family specified is **AF_UNSPEC**, the system attempts to create sockets in all of the supported address families (**AF_LOCAL** and **AF_TIPC**). The socket addresses used for the server's sockets are selected automatically, and cannot be specified by the server application with **salCreate()**.

A server can be accessed by clients within the scope that is defined when the server is created with the **salCreate()** routine. The scope can have the following values: **private**, **node**, or **cluster**. These values designate an access scope for limiting access.
144

to the same task (kernel or process), the same node (the kernel and all processes on
that node), and a set of nodes, respectively. For example, the following call would
create a socket named **foo** with **cluster** scope:

```
salCreate("foo@cluster",1,5)
```

Once created, a SAL server must be configured with one or more processing
routines before it is activated. These routines can be configured by calling
**salServerRtnSet()**.

Once the server is ready, **salRun()** is called to start the server activities. The
**salRun()** routine never returns unless there is an error or one of the server
processing routines requests it. You must call **salDelete()** to delete the server and
its sockets regardless of whether or not the routine has terminated. This is
accomplished with **salDelete()**. This routine can be called only by tasks in the
process (or the kernel) where the server was created. In order for tasks outside the
process to remove a service name from SNS, **salRemove()** must be used. The
**salRemove()** routine does not close sockets, nor does it delete the server. It only
deletes the SNS entry, and therefore access to any potential clients.

For more information, including sample service code, see the VxWorks API
reference for the **salServer** library.

**SAL Client Library**

The SAL client library provides a simple means for implementing a socket-based
client application. The data structures and routines provided by SAL allow the
application to easily communicate with socket-based server applications that are
registered with the Socket Name Service (see **Socket Name Service**, p.138).
Additional routines can be used to communicate with server applications that are
not registered with the SNS. The SAL client library is made of the following
routines:

**salOpen()**
  Establishes communication with a named socket-based server.

**salSocketFind()**
  Finds sockets for a named socket-based server.

**salNameFind()**
  Finds services with the specified name.

**salCall()**
  Invokes a socket-based server.

A client application typically calls **salOpen()** to create a client socket and connect
it to the named server application. The client application can then communicate
with the server by passing the socket descriptor to standard socket API routines, such as `send()` and `recv()`.

As an alternative, the client application can perform a `send()` and `recv()` as a single operation using `salCall()`. When the client application no longer needs to communicate with a server it calls the standard socket `close()` routine to close the socket to the server.

A client socket can be shared between two or more tasks. In this case, however, special care must be taken to ensure that a reply returned by the server application is handled by the correct task.

The `salNameFind()` and `salSocketFind()` routines facilitate the search of the server and provide more flexibility for the client application.

The `salNameFind()` routine provides a lookup mechanism for services based on pattern matching, which can be used with (multiple) wild cards to locate similar names. For example, if the names are `foo`, `foo2`, and `foobar`, then a search using `foo*` would return them all. The scope of the search can also be specified. For example, a client might want to find any server up to a given scope, or only within a given scope. In the former case the `upto_` prefix can be added to the scope specification. For example, `upto_node` defines a search that look for services in all processes and in the kernel in a node.

Once a service is found, the `salSocketFind()` routine can be used to return the proper socket ID. This can be useful if the service has multiple sockets, and the client requires use of a specific one. This routine can also be used with wild cards, in which case the first matching server socket is returned.

For more information, including sample client code, see the VxWorks API reference for the `salClient` library.

### Configuring VxWorks for Message Channels

To provide the full set of message channel facilities in a system, configure VxWorks with the following components:

- `INCLUDE_UN_COMP`
- `INCLUDE_DSI_POOL`
- `INCLUDE_DSI_SOCKET`
- `INCLUDE_SAL_SERVER`
- `INCLUDE_SAL_CLIENT`

Note that `INCLUDE_UN_COMP` is required for both single and multi-node systems, as it provides support for communication between SAL and SNS.
While COMP provides for standard socket support, it has no dependency on TCP/IP networking facilities, which can be left out of a system if they are not otherwise needed.

For multi-node systems, TIPC components must also be included. See the Wind River TIPC Programmer’s Guide for more information.

**SNS Configuration**

In addition to the COMP, DSI, and SAL components, one of the four following components is required for SNS:

- **INCLUDE_SNS** to run SNS as a kernel daemon.
- **INCLUDE_SNS_RTP** to start SNS as a process automatically at boot time.
- **INCLUDE_SNS_MP** to run SNS as a kernel daemon supporting distributed named sockets.
- **INCLUDE_SNS_MP_RTP** to start SNS as a process automatically at boot time supporting distributed named sockets.

Note that including a distributed SNS server will cause the inclusion of TIPC which in turn will force the inclusion of other networking components.

**Running SNS as a Process**

In order to run SNS as a process (RTP), the developer must also build the server, add it to ROMFS, configure VxWorks with ROMFS support, and then rebuild the entire system:

a. Build `installDir/vxworks-6.x/target/usr/apps/dsi/snsd/snsd.c` (using the makefile in the same directory) to create `snsServer.vxe`.

b. Copy `snsServer.vxe` to the ROMFS directory (creating the directory first, if necessary).

The **INCLUDE_SNS_RTP** and **INCLUDE_SNS_MP_RTP** components must know the location of the server in order to start it at boot time. They expect to find the server in the ROMFS directory. If you wish to store the server somewhere else (in another file system to reduce the VxWorks image size, for example) use the **SNS_PATHNAME** parameter to identify the location.

c. Configure VxWorks with the ROMFS component.

d. Rebuild VxWorks.
These steps can also be performed with Wind River Workbench (see the Wind River Workbench User’s Guide). For information about ROMFS, see 7.8 Read-Only Memory File System: ROMFS, p.323.

SNS Configuration

The following SNS component parameters can usually be used without modification:

SNS_LISTEN_BACKLOG
This parameter defines the number of outstanding service requests that the SNS server can track on the socket that it uses to service SNS requests from SAL routines. The default value is 5. The value may be increased if some SAL requests are not processed on a busy system.

SNS_DISTRIBUTED_SERVER_TYPE and SNS_DISTRIBUTED_SERVER_INSTANCE
These parameters are used in the multi-node configuration of SNS servers to define the TIPC port name that all SNS servers use. The default is type 51 and instance 51 in the TIPC name tables. If this type and instance conflict with other usages in the network, they can be changed to values that are unique for the network. Note that it is recommended to use a type of 50 or above (types 0 through 7 are reserved by TIPC).

The SNS server creates a COMP socket for local communication with the socket address of 0x0405. All of the SAL routines send messages to the SNS server at this socket address.

⚠️ CAUTION: It is recommended that you do not change the default values of the SNS_PRIORITY and SNS_STACK_SIZE parameters. The default for SNS_PRIORITY is 50 and the default for SNS_STACK_SIZE is 20000.

Show Routines

The show routines related to COMP can be included by adding the INCLUDE_UN COMP_SHOW component. The snsShow() routine is included with the INCLUDE_SNS_SHOW component. In order to use netstat() the network show routines must be included. Note that this will force the inclusion of networking components.

For information about processes and applications, see 2. Applications and Processes.
Comparing Message Channels and Message Queues

Message channels can be used similarly to message queues, to exchange data between two tasks. Both methods allow multiple tasks to send and receive from the same channel. The main differences between these two mechanisms are:

- Message channels can be used to communicate between nodes, but message queues cannot.
- Message channels are connection-oriented while message queues are not. There is no way to establish a connection between two tasks with message queues. In a connection-oriented communication, the two end-points are aware of each other, and if one leaves the other eventually finds out. By way of analogy, a connection-oriented communication is like a telephone call, whereas a connection-less communication is like sending a letter. Both models are valid, and the requirements of the application should determine their use.

Each message queue is unidirectional. In order to establish a bidirectional communication, two queues are needed, one for each end-point (see Figure 3-14). Each message channel is bidirectional and data can be sent from both end-points at any time. That is, each message channel provides connection-oriented full-duplex communication.

- The messages communicated by message channels can be of variable size, whereas those communicated by message queues have a maximum size that is defined when the queue is created. Message channels therefore allow for a better utilization of system resources by using exactly what is needed for the message, and nothing more.
- Message queues have a fixed capacity. Only a pre-defined number of messages can be in a queue at any one time. Message channels, on the other hand, have a flexible capacity. There is no limit to the number of messages that a message channel can handle.
- Message channels provide location transparency. An endpoint can be referred to by a name, that is by a simple string of characters (but a specific address can also be used). Message queues only provide location transparency for interprocess communication when they are created as public objects.
- Message channels provide a simple interface for implementing a client/server paradigm. A location transparent connection can be established by using two simple calls, one for the client and one for the server. Message queues do not provide support for client/server applications.
- Message channels use the standard socket interface and support the `select()` routine; message queues do not.
3.3.9 Network Communication

To communicate peer on a remote networked system, you can use an Internet domain socket or RPC. For information on working with Internet domain sockets under VxWorks, see the Wind River Network Stack for VxWorks 6 Programmer’s Guide: Sockets under the Wind River Network Stack. For information on RPC, see Wind River Network Stack for VxWorks 6 Programmer’s Guide: RPC Components and the VxWorks API reference for rpcLib.

3.3.10 Signals

Signals are an operating system facility designed for handling exceptional conditions and asynchronously altering the flow of control. In many respects signals are the software equivalent to hardware interrupts. Signals generated by the operating system include those produced in response to bus errors and floating point exceptions. The signal facility also provides APIs that can be used to generate and manage signals programmatically.

In applications, signals are most appropriate for error and exception handling, and not for a general-purpose inter-task communication. Common uses include using signals to kill processes and tasks, to send signal events when a timer has fired or message has arrived at a message queue, and so on.

In accordance with POSIX, VxWorks supports 63 signals, each of which has a unique number and default action (defined in signal.h). The value 0 is reserved for use as the NULL signal.

Signals can be raised (sent) from tasks to tasks or to processes. Signals can be either caught (received) or ignored by the receiving task or process. Whether signals are caught or ignored generally depends on the setting of a signal mask. In the kernel, signal masks are specific to tasks, and if no task is set up to receive a specific signal, it is ignored. In user space, signal masks are specific to processes; and some signals, such as SIGKILL and SIGSTOP, cannot be ignored.
To manage responses to signals, you can create and register signal handling routines that allow a task to respond to a specific signal in whatever way is useful for your application.

A user-space (process-based) task can raise a signal for any of the following:

- itself
- any other task in its process
- any public task in the system
- its own process
- any other process in the system

A user-space task cannot raise a signal for a kernel task (even if it is a public task; for information about public tasks, see *Task Names and IDs*, p.91.). By default, signals sent to a task in a process results in the termination of the process.

Unlike kernel signal generation and delivery, which runs in the context of the task or ISR that generates the signal, user-space signal generation is performed by the sender task, but the signal delivery actions take place in the context of the receiving task.

For processes, signal handling routines apply to the entire process, and are not specific to any one task in the process.

Signal handling is done on an process-wide basis. That is, if a signal handler is registered by a task, and that task is waiting for the signal, then a signal to the process is handled by that task. Otherwise, any task that does not have the signal blocked will handle the signal. If there is no task waiting for a given signal, the signal remains pended in the process until a task that can receive the signal becomes available.

Each task has a signal mask associated with it. The signal mask determines which signals the task accepts. When a task is created, its signal mask is inherited from the task that created it. If the parent is a kernel task (that is, if the process is spawned from the kernel), the signal mask is initialized with all signals unblocked. It also inherits default actions associated with each signal. Both can later be changed with `sigprocmask()`.

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

The following non-POSIX APIs are also provided for signals: `taskSigqueue()`, `taskKill()`, `rtpKill()`, and `taskRaise()`. These APIs are provided to facilitate porting VxWorks kernel applications to RTP applications.
NOTE: POSIX signals are handled differently in the kernel and in real-time processes. In the kernel the target of a signal is always a task; but in user space, the target of a signal may be either a specific task or an entire process.

NOTE: The VxWorks implementation of sigLib does not impose any special restrictions on operations on SIGKILL, SIGCONT, and SIGSTOP signals such as those imposed by UNIX. For example, the UNIX implementation of signal() cannot be called on SIGKILL and SIGSTOP.

For information about using signals in the kernel, see VxWorks Kernel Programmer’s Guide: Multitasking.

In addition to signals, VxWorks also provides another type of event notification with the VxWorks events facility. While signal events are completely asynchronous, VxWorks events are sent asynchronously, but received synchronously, and do not require a signal handler. For more information, see 3.3.7 VxWorks Events, p.126.

Configuring VxWorks for Signals

By default, VxWorks includes the basic signal facility component INCLUDE_SIGNALS. This component automatically initializes signals with sigInit().

To use the signal event facility, configure VxWorks with the INCLUDE_SIGEVENT component. Note that SIGEV_THREAD option is only supported in processes, and that it requires that VxWorks also be configured with the INCLUDE_SIGEVENTS_THREAD component and full POSIX thread support (the BUNDLE_RTP_POSIX_PSE52 bundle includes everything required for this option).

The maximum number of queued signals in a process is set with the configuration parameter RTP_SIGNAL_QUEUE_SIZE. The default value, 32, is set in concordance with the POSIX 1003.1 standard (_POSIX_SIGQUEUE_MAX). Changing the value to a lower number may cause problems for applications relying on POSIX guidelines.

Process-based (RTP) applications are automatically linked with the sigLib library when they are compiled. Initialization of the library is automatic when the process starts.
Basic Signal Routines

Signals are in many ways analogous to hardware interrupts. The basic signal facility provides a set of 63 distinct signals. A signal handler binds to a particular signal with `sigvec()` or `sigaction()` in much the same way that an ISR is connected to an interrupt vector with `intConnect()`. A signal can be asserted by calling `kill()` or `sigqueue()`. This is similar to the occurrence of an interrupt. The `sigprocmask()` routine lets signals be selectively inhibited. Certain signals are associated with hardware exceptions. For example, bus errors, illegal instructions, and floating-point exceptions raise specific signals.

VxWorks also provides a POSIX and BSD-like `kill()` routine, which sends a signal to a task.

For a list and description of the basic set of POSIX signal routines provided by VxWorks for use with processes, see Table 3-17.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>signal()</code></td>
<td>Specifies the handler associated with a signal.</td>
</tr>
<tr>
<td><code>kill()</code></td>
<td>Sends a signal to a process.</td>
</tr>
<tr>
<td><code>raise()</code></td>
<td>Sends a signal to the caller’s process.</td>
</tr>
<tr>
<td><code>sigaction()</code></td>
<td>Examines or sets the signal handler for a signal.</td>
</tr>
<tr>
<td><code>sigsuspend()</code></td>
<td>Suspends a task until a signal is delivered.</td>
</tr>
<tr>
<td><code>sigpending()</code></td>
<td>Retrieves a set of pending signals blocked from delivery.</td>
</tr>
<tr>
<td><code>sigemptyset()</code></td>
<td>Manipulates a signal mask.</td>
</tr>
<tr>
<td><code>sigfillset()</code></td>
<td></td>
</tr>
<tr>
<td><code>sigaddset()</code></td>
<td></td>
</tr>
<tr>
<td><code>sigdelset()</code></td>
<td></td>
</tr>
<tr>
<td><code>sigismember()</code></td>
<td></td>
</tr>
<tr>
<td><code>sigprocmask()</code></td>
<td>Sets the mask of blocked signals.</td>
</tr>
<tr>
<td><code>sigprocmask()</code></td>
<td>Adds to a set of blocked signals.</td>
</tr>
<tr>
<td><code>sigaltstack()</code></td>
<td>Set or get a signal’s alternate stack context.</td>
</tr>
</tbody>
</table>
For more information about signal routines, see the VxWorks API reference for sigLib.

Queued Signal Routines

The `sigqueue()` family of routines provides an alternative to the `kill()` family of routines for sending signals. The important differences between the two are the following:

- The `sigqueue()` routine includes an application-specified value that is sent as part of the signal. This value supplies whatever context is appropriate for the signal handler. This value is of type `sigval` (defined in `signal.h`); the signal handler finds it in the `si_value` field of one of its arguments, a structure `siginfo_t`.

- The `sigqueue()` routine enables the queueing of multiple signals for any task. The `kill()` routine, by contrast, delivers only a single signal, even if multiple signals arrive before the handler runs.

VxWorks includes signals reserved for application use, numbered consecutively from `SIGRTMIN` to `SIGRTMAX`. The number of signals reserved is governed by the `RTSIG_MAX` macro (with a value of 16), which defined in the POSIX 1003.1 standard. The signal values themselves are not specified by POSIX. For portability, specify these signals as offsets from `SIGRTMIN` (for example, use `SIGRTMIN+2` to refer to the third reserved signal number). All signals delivered with `sigqueue()` are queued by numeric order, with lower-numbered signals queuing ahead of higher-numbered signals.

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

The basic queued signal routines are described in Table 3-18. For detailed information on signals, see the API reference for `sigLib`. 
Additional non-POSIX VxWorks queued signal routines are described in Table 3-19. These routines are provided for assisting in porting VxWorks 5.x kernel applications to processes. The POSIX routines described in Table 3-18 should be used for developing new applications that execute as real-time processes.

Note that a parallel set of non-POSIX APIs are provided for the kill( ) family of POSIX routines—taskKill( ), rtpKill( ), and rtpTaskKill( ).

Table 3-18  POSIX 1003.1b Queued Signal Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigqueue( )</td>
<td>Sends a queued signal to a process.</td>
</tr>
<tr>
<td>sigwaitinfo( )</td>
<td>Waits for a signal.</td>
</tr>
<tr>
<td>sigtimedwait( )</td>
<td>Waits for a signal with a timeout.</td>
</tr>
</tbody>
</table>

Table 3-19  Non-POSIX Queued Signal Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskSigqueue( )</td>
<td>Sends a queued signal from a task in a process to another task in the same process, to a public task in another process, or from a kernel task to a process task.</td>
</tr>
<tr>
<td>rtpSigqueue( )</td>
<td>Sends a queued signal from a kernel task to a process or from a process to another process.</td>
</tr>
<tr>
<td>rtpTaskSigqueue( )</td>
<td>Sends a queued signal from a kernel task to a specified task in a process (kernel-space only).</td>
</tr>
</tbody>
</table>
Example 3-4  Queued Signals

```c
#include <stdio.h>
#include <signal.h>
#include <taskLib.h>
#include <rtpLib.h>
#ifdef _WRS_KERNEL
#include <private/rtpLibP.h>
#include <private/taskLibP.h>
#include <errnoLib.h>
#endif

typedef void (*FPTR) (int);

void sigMasterHandler
(
    int     sig,                /* caught signal */
#ifdef _WRS_KERNEL
    int      code,              /* signal info */
#else
    siginfo_t * pInfo,          /* signal info */
#endif
    struct sigcontext *pContext /* unused */
);

****************************************************************************
*
* main - entry point for the queued signal demo
*
* This routine acts the task entry point in the case of the demo spawned as a
* kernel task. It also can act as a RTP entry point in the case of RTP based
* demo.
*/

STATUS main (void)
{
    sigset_t sig = sigmask (SIGUSR1);
    union sigval sval;
    struct sigaction in;
    sigprocmask (SIG_UNBLOCK, &sig, NULL);

    in.sa_handler = (FPTR) sigMasterHandler;
    in.sa_flags = 0;
    (void) sigemptyset (&in.sa_mask);

    if (sigaction (SIGUSR1, &in, NULL) != OK)
    {
        printf ("Unable to set up handler for task (0x%x)\n", taskIdCurrent);
        return (ERROR);
    }

    printf ("Task 0x%x installed signal handler for signal # %d.\n READY for signal.\n", taskIdCurrent, SIGUSR1);
}
for (;;) {
}

/**************************************************************************
* sigMasterHandler - signal handler
* This routine is the signal handler for the SIGUSR1 signal
*/

void sigMasterHandler
(
    int     sig,                /* caught signal */
    #ifdef _WRS_KERNEL
    int     code,
    #else
    siginfo_t * pInfo ,         /* signal info */
    #endif
    struct sigcontext *pContext /* unused */
){
    printf ("Task 0x%x got signal # %d  signal value %d \n",
        taskIdCurrent, sig,
        #ifdef _WRS_KERNEL
        code
        #else
        pInfo->si_value.sival_int
        #endif
    );
}

/**************************************************************************
* sig - helper routine to send a queued signal
* This routine can send a queued signal to a kernel task or RTP task or RTP.
* <id> is the ID of the receiver entity. <value> is the value to be sent
* along with the signal. The signal number being sent is SIGUSR1.
*/

#ifdef _WRS_KERNEL
STATUS sig
(
    int id,
    int val
)
{
    union sigval        valueCode;

    valueCode.sival_int = val;
}
if (TASK_ID_VERIFY (id) == OK)
{
if (IS_KERNEL_TASK (id))
{
if (sigqueue (id, SIGUSR1, valueCode) == ERROR)
{
printf ("Unable to send SIGUSR1 signal to 0x%x, errno = 0x%x\n", id, errnoGet());
return ERROR;
}
}
else
{
    rtpTaskSigqueue ((WIND_TCB *)id, SIGUSR1, valueCode);
}
else if (OBJ_VERIFY ((RTP_ID)id, rtpClassId) != ERROR)
{
    rtpSigqueue ((RTP_ID)id, SIGUSR1, valueCode);
} else
{
    return (ERROR);
}
return (OK);
#endif
The code provided in this example can be used to do any of the following:

- Send a queued signal to a kernel task.
- Send a queued signal to a task in a process (RTP).
- Send a queued signal to a process.

The \texttt{sig()} routine provided in this code is a helper routine used to send a queued signal.

To use the code as an RTP application, VxWorks must be configured with \texttt{BUNDLE_NET_SHELL} and \texttt{BUNDLE_RTP_POSIX_PSE52}.

To send a queued signal to a process:

1. Build the application as an RTP application.
2. Spawn the application. For example:

   \begin{verbatim}
   -> rtpSp "signal_ex.vxe"
   value = 10531472 = 0xa0b290
   \end{verbatim}

   -> Task 0x10000 installed signal handler for signal # 30. Ready for signal.

3. Determine the RTP ID; for example:

   \begin{verbatim}
   -> rtpShow
   \end{verbatim}

   \begin{verbatim}
   NAME         ID          STATE      ENTRY ADDR  OPTIONS   TASK CNT
   -------------------------- --------------- ---------- ---------- --------
   signal_ex.vxe  0xa0b290  STATE_NORMAL  0x10000298  0x1        1
   \end{verbatim}

   value = 1 = 0x1

4. From a kernel task (note that the kernel shell runs as a kernel task), use the \texttt{sig()} helper routine to send a queued signal to an RTP, where the \texttt{id} parameter is the RTP ID. For example, using the RTP ID of 0xa0b290 found in the previous step:

   \begin{verbatim}
   -> sig 0xa0b290, 50
   value = 0 = 0x0
   \end{verbatim}

   -> Task 0x10000 got signal # 30 signal value 50

To send a queued signal to a task in a process:

1. Build the code as an RTP application.
2. Spawn the application.
3. From a kernel task (such as the kernel shell), use the \texttt{sig()} helper routine to send a queued signal to an RTP task, where the \texttt{id} parameter is the RTP task ID.

For information on using the code in the kernel (as a kernel application), see the \textit{VxWorks Application Programmer's Guide: Multitasking}.
Signal Events

The signal event facility allows a pthread or task to receive notification that a particular event has occurred (such as the arrival of a message at a message queue, or the firing of a timer) by way of a signal.

The following routines can be used to register for signal notification of their respective event activities: `mq_notify()`, `timer_create()`, `timer_open()`, `aio_read()`, `aio_write()` and `lio_listio()`.

The POSIX 1003.1-2001 standard defines three signal event notification types:

**SIGEV_NONE**
Indicates that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion.

**SIGEV_SIGNAL**
Indicates that a signal is generated when the event occurs.

**SIGEV_THREAD**
Provides for callback functions for asynchronous notifications done by a function call within the context of a new thread. This provides a multi-threaded process with a more natural means of notification than signals. VxWorks supports this option in user space (processes), but not in the kernel.

The notification type is specified using the `sigevent` structure, which is defined in `installDir/vxworks-6.x/target/h/sigeventCommon.h`. A pointer the structure is used in the call to register for signal notification; for example, with `mq_notify()`.

To use the signal event facility, configure VxWorks with the `INCLUDE_SIGEVENT` component.

As noted above, the `SIGEV_THREAD` option is only supported in processes, and it requires that VxWorks be configured with the `INCLUDE_SIGEVENTS_THREAD` component and full POSIX thread support (the `BUNDLE_RTP_POSIX_PSE52` bundle includes everything required for this option).

Signal Handlers

Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism. And normally, signal handlers should be treated like ISRs: no routine should be called from a signal handler that might cause the handler to block. Because signals are asynchronous, it is difficult to predict which resources might be unavailable when a particular
signal is raised.

To be perfectly safe, call only those routines listed in Table 3-20. Deviate from this practice only if you are certain that your signal handler cannot create a deadlock situation.

In addition, you should be particularly careful when using C++ for a signal handler or when invoking a C++ method from a signal handler written in C or assembly. Some of the issues involved in using C++ include the following:

- The VxWorks `intConnect()` and `signal()` routines require the address of the function to execute when the interrupt or signal occurs, but the address of a non-static member function cannot be used, so static member functions must be implemented.
- Objects cannot be instantiated or deleted in signal handling code.
- C++ code used to execute in a signal handler should restrict itself to Embedded C++. No exceptions nor run-time type identification (RTTI) should be used.

### Table 3-20 Routines Callable by Signal Handlers

<table>
<thead>
<tr>
<th>Library</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLib</td>
<td>All routines</td>
</tr>
<tr>
<td>errnoLib</td>
<td><code>errnoGet()</code>, <code>errnoSet()</code></td>
</tr>
<tr>
<td>eventLib</td>
<td><code>eventSend()</code></td>
</tr>
<tr>
<td>logLib</td>
<td><code>logMsg()</code></td>
</tr>
<tr>
<td>lstLib</td>
<td>All routines except <code>lstFree()</code></td>
</tr>
<tr>
<td>msgQLib</td>
<td><code>msgQSend()</code></td>
</tr>
<tr>
<td>rngLib</td>
<td>All routines except <code>rngCreate()</code> and <code>rngDelete()</code></td>
</tr>
<tr>
<td>semLib</td>
<td><code>semGive()</code> except mutual-exclusion semaphores, <code>semFlush()</code></td>
</tr>
<tr>
<td>sigLib</td>
<td><code>kill()</code></td>
</tr>
<tr>
<td>taskLib</td>
<td><code>taskSuspend()</code>, <code>taskResume()</code>, <code>taskPrioritySet()</code>, <code>taskPriorityGet()</code>, <code>taskIdVerify()</code>, <code>taskIdDefault()</code>, <code>taskIsReady()</code>, <code>taskIsSuspended()</code>, <code>taskTcb()</code></td>
</tr>
<tr>
<td>tickLib</td>
<td><code>tickAnnounce()</code>, <code>tickSet()</code>, <code>tickGet()</code></td>
</tr>
</tbody>
</table>
Most signals are delivered asynchronously to the execution of a program. Therefore programs must be written to account for the unexpected occurrence of signals, and handle them gracefully. Unlike ISR’s, signal handlers execute in the context of the interrupted task.

VxWorks does not distinguish between normal task execution and a signal context, as it distinguishes between a task context and an ISR. Therefore the system has no way of distinguishing between a task execution context and a task executing a signal handler. To the system, they are the same.

When you write signal handlers make sure that they:

- Release resources prior to exiting:
  - Free any allocated memory.
  - Close any open files.
  - Release any mutual exclusion resources such as semaphores.

- Leave any modified data structures in a sane state.

Notify the parent process with an appropriate error return value. Mutual exclusion between signal handlers and tasks must be managed with care. In general, users should avoid the following activity in signal handlers:

- Taking mutual exclusion (such as semaphores) resources that can also be taken by any other element of the application code. This can lead to deadlock.

- Modifying any shared data memory that may have been in the process of modification by any other element of the application code when the signal was delivered. This compromises mutual exclusion and leads to data corruption.

- Using `longjmp()` to change the flow of task execution. If `longjmp()` is used in a signal handler to re-initialize a running task, you must ensure that the signal is not sent to the task while the task is holding a critical resource (such as a kernel mutex). For example, if a signal is sent to a task that is executing `malloc()`, the signal handler that calls `longjmp()` could leave the kernel in an inconsistent state.

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

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

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

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

### 3.4 Timers

VxWorks provides watchdog timers, but they can only be used in the kernel (see VxWorks Kernel Programmer's Guide: Multitasking). However, process-based applications can use POSIX timers (see 4.8 POSIX Clocks and Timers, p.177).
4
POSIX Facilities

4.1 Introduction 164
4.2 Configuring VxWorks with POSIX Facilities 166
4.3 General POSIX Support 169
4.4 Standard C Library: Libc 171
4.6 POSIX Namespace 174
4.7 POSIX Process Support 176
4.8 POSIX Clocks and Timers 177
4.9 POSIX Asynchronous I/O 181
4.10 POSIX Advisory File Locking 181
4.11 POSIX Page-Locking Interface 182
4.12 POSIX Threads 183
4.13 POSIX Thread Mutexes and Condition Variables 192
4.14 POSIX and VxWorks Scheduling 197
4.15 POSIX Semaphores 208
4.16 POSIX Message Queues 218
4.17 POSIX Signals 232
4.18 POSIX Memory Management 232
4.1 Introduction

The VxWorks user space environment provides the execution environment of choice for POSIX applications. The compliance to the POSIX standard is very high for all the APIs and behaviors expected by the PSE52 profile (Realtime Controller System Profile), which is described by IEEE Std 1003.13-2003 (POSIX.13). Various APIs which operate on a task in kernel mode, operate on a process in user mode—such as `kill()`, `exit()`, and so on.

Support for a conforming application is achieved in a process (RTP) in the following way:

- By configuring the VxWorks kernel with the appropriate functionality, such as a POSIX-compliant file system and scheduler.
- By using only POSIX PSE52 profile APIs in the application.

This implementation of the POSIX.13 PSE52 profile is based on IEEE Std 1003.1-2001 (POSIX.1). As defined by the PSE52 profile, it is restricted to individual processes.

VxWorks does provide a process model that allows multiple processes, and provides many POSIX.1 APIs associated with facilities such as networking and inter-process communication that are not part of the PSE52 profile.

Note that the only VxWorks file system that is POSIX-compliant is the Highly Reliable File System (for more information, see 7.4 Highly Reliable File System: HRFS, p.297).

For detailed information about POSIX standards and facilities, see The Open Group Web sites at http://www.opengroup.org/ and http://www.unix.org/.

POSIX and Real-Time Systems

While VxWorks provides many POSIX compliant APIs, not all POSIX APIs are suitable for embedded and real-time systems, or are entirely compatible with the VxWorks operating system architecture. In a few cases, therefore, Wind River has imposed minor limitations on POSIX functionality to serve either real-time systems or VxWorks compatibility. For example:
• Swapping memory to disk is not appropriate in real-time systems, and VxWorks provides no facilities for doing so. It does, however, provide POSIX page-locking routines to facilitate porting code to VxWorks. The routines otherwise provide no useful function—pages are always locked in VxWorks systems (for more information see 4.11 POSIX Page-Locking Interface, p.182).

• VxWorks tasks are scheduled on a system-wide basis; processes themselves cannot be scheduled. As a consequence, while POSIX access routines allow two values for contention scope (PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS), only system-wide scope is implemented in VxWorks for these routines (for more information, see 4.12 POSIX Threads, p.183 and 4.14 POSIX and VxWorks Scheduling, p.197).

Any such limitations on POSIX functionality are identified in this chapter, or in other chapters of this guide that provide more detailed information on specific POSIX APIs.

**POSIX and VxWorks Facilities**

This chapter describes the POSIX support provided by VxWorks and VxWorks-specific POSIX extensions. In addition, it compares native VxWorks facilities with similar POSIX facilities that are also available with VxWorks.

The qualifier *VxWorks* is used in this chapter to identify native non-POSIX APIs for purposes of comparison with POSIX APIs. For example, you can find a discussion of VxWorks semaphores contrasted to POSIX semaphores in 4.15.1 Comparison of POSIX and VxWorks Semaphores, p.209, although POSIX semaphores are also implemented in VxWorks.

VxWorks extensions to POSIX are identified as such.

---

**NOTE:** This chapter provides information about POSIX facilities available for real-time processes (RTPs). For information about facilities available in the VxWorks kernel, see the corresponding chapter in the VxWorks Kernel Programmer’s Guide.
4.2 Configuring VxWorks with POSIX Facilities

Process-based (RTP) applications are automatically linked with the appropriate user-level POSIX libraries when they are compiled. The libraries are automatically initialized at run time.

Some user-level POSIX features also require support from the kernel. If VxWorks is not configured with the POSIX components required by an application using these features, the calls that lack support return an ENOSYS error at run-time.

Note that support for POSIX threads (pthreads) in processes requires that the kernel be configured with the POSIX thread scheduler component INCLUDE_POSIX_PTHREAD_SCHEDULER.

⚠️ CAUTION: The set of components used for POSIX support in user space is not the same as the set of components used for POSIX support in kernel space. For information about the components for kernel space, see Table 4-1 and the VxWorks Kernel Programmer’s Guide: POSIX Facilities for the appropriate component bundle.

4.2.1 POSIX PSE52 Support

The components required for POSIX PSE53 support for processes are provided with the BUNDLE_RTP_POSIX_PSE52 component bundle. If memory constraints require a finer-grained configuration, individual components can be used for selected features. See the configuration instructions for individual POSIX features throughout this chapter for more information in this regard.

If an application requires a PSE52 file system, then VxWorks must be configured with the following components as well:

- INCLUDE_HRFS for the highly reliable file system. See 7.4 Highly Reliable File System: HRFS, p.297.
- The appropriate device driver component — for example INCLUDE_ATA or INCLUDE_XBD_RAMDRV—and INCLUDE_XBD_BLK_DEV if the device driver requires it (that is, if it is not XBD-compatible). See the VxWorks Kernel Programmer’s Guide: I/O System.

In addition, a /tmp directory must be created at run-time and must appear on the virtual root file system, which is provided with the BUNDLE_RTP_POSIX_PSE52 component bundle (see 7.4.2 HRFS and POSIX PSE52, p.298).

VxWorks PSE52 support in processes includes the following units of functionality (POSIX.13):
4.2 Configuring VxWorks with POSIX Facilities

- POSIX_C_LANG_JUMP
- POSIX_C_LANG_MATH
- POSIX_C_LANG_SUPPORT
- POSIX_DEVICE_IO
- POSIX_FD_MGMT
- POSIX_FILE_LOCKING
- POSIX_FILE_SYSTEM
- POSIX_SIGNALS
- POSIX_SINGLE_PROCESS
- POSIX_THREADS_BASE
- XSI_THREAD_MUTEX_EXT
- XSI_THREADS_EXT

It also provides the following options (POSIX.1):

- _POSIX_CLOCK_SELECTION
- _POSIX_FSYNC
- _POSIX_MAPPED_FILES
- _POSIX_MEMLOCK
- _POSIX_MEMLOCK_RANGE
- _POSIX_MESSAGE_PASSING
- _POSIX_MONOTONIC_CLOCK
- _POSIX_NO_TRUNC
- _POSIX_REALTIME_SIGNALS
- _POSIX_SHARED_MEMORY_OBJECTS
- _POSIX_SYNCHRONIZED_IO
- _POSIX_THREAD_ATTR_STACKADDR
- _POSIX_THREAD_ATTR_STACKSIZE
- _POSIX_THREAD_CPUTIME
- _POSIX_THREAD_PRIO_INHERIT
- _POSIX_THREAD_PRIO_PROTECT
- _POSIX_THREAD_PRIORITY_SCHEDULING
- _POSIX_THREAD_SAFE_FUNCTIONS
- _POSIX_THREAD_SPOORADIC_SERVER
- _POSIX_THREADS
- _POSIX_TIMEOUTS
- _POSIX_TIMERS

For information about POSIX namespace isolation for PSE52-conforming applications, see 4.6 POSIX Namespace, p.174.

⚠️ CAUTION: If a PSE52-conforming application relies on C99 constructs, the Wind River Compiler must be used. The version of the GNU compiler provided with this release does not support the C99 language constructs.
4.2.2 VxWorks Components for POSIX Facilities

Table 4-1 provides an overview of the individual VxWorks components that must be configured in the kernel to provide support for the specified POSIX facilities.

<table>
<thead>
<tr>
<th>POSIX Facility</th>
<th>for Kernel</th>
<th>Required VxWorks Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard C library</td>
<td>INCLUDE_ANSI_* components</td>
<td>Dinkum C library (libc)</td>
</tr>
<tr>
<td>Asynchronous I/O with system driver</td>
<td>INCLUDE_POSIX_AIO, INCLUDE_POSIX_AIO_SYSDR V and INCLUDE_PIPES</td>
<td>INCLUDE_POSIX_CLOCKS and INCLUDE_POSIX_TIMERS</td>
</tr>
<tr>
<td>Clocks</td>
<td>INCLUDE_POSIX_CLOCKS</td>
<td>INCLUDE_POSIX_CLOCKS</td>
</tr>
<tr>
<td>Directory and file utilities</td>
<td>INCLUDE_POSIX_DIRLIB</td>
<td>N/A</td>
</tr>
<tr>
<td>ftruncate()</td>
<td>INCLUDE_POSIX_FTRUNC</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory locking</td>
<td>INCLUDE_POSIX_MEM</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory management</td>
<td>N/A</td>
<td>INCLUDE_RTP</td>
</tr>
<tr>
<td>Memory-mapped files</td>
<td>N/A</td>
<td>INCLUDE_POSIX_MAPPED_FILES</td>
</tr>
<tr>
<td>Shared memory objects</td>
<td>N/A</td>
<td>INCLUDE_POSIX_MAPPED_FILES and INCLUDE_POSIX_SHM</td>
</tr>
<tr>
<td>Message queues</td>
<td>INCLUDE_POSIX_MQ</td>
<td>INCLUDE_POSIX_MQ</td>
</tr>
<tr>
<td>pthreads</td>
<td>INCLUDE_POSIX_THREADS</td>
<td>INCLUDE_POSIX_CLOCKS, INCLUDE_POSIX_PTHREAD_SCHEDULE, and INCLUDE_PTHREAD_CPUTIME</td>
</tr>
<tr>
<td>Process Scheduling API</td>
<td>INCLUDE_POSIX_SCHED</td>
<td>N/A</td>
</tr>
<tr>
<td>Semaphores</td>
<td>INCLUDE_POSIX_SEM</td>
<td>INCLUDE_POSIX_SEM</td>
</tr>
<tr>
<td>Signals</td>
<td>INCLUDE_POSIX_SIGNALS</td>
<td>N/A</td>
</tr>
</tbody>
</table>
4 POSIX Facilities

4.3 General POSIX Support

Many POSIX-compliant libraries are provided for VxWorks. These libraries are listed in Table 4-2; see the API references for these libraries for detailed information.

Networking facilities are described in the Wind River Network Stack for VxWorks 6 Programmer’s Guide.

<table>
<thead>
<tr>
<th>POSIX Facility</th>
<th>Required VxWorks Component for Kernel</th>
<th>Required VxWorks Component for Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timers</td>
<td>INCLUDE_POSIX_TIMERS</td>
<td>INCLUDE_POSIX_TIMERS</td>
</tr>
<tr>
<td>Trace</td>
<td>N/A</td>
<td>INCLUDE_POSIX_TRACE</td>
</tr>
<tr>
<td>POSIX PSE52</td>
<td>N/A</td>
<td>BUNDLE_RTP_POSIX_PSE52</td>
</tr>
<tr>
<td>support</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following sections of this chapter describe the POSIX APIs available to user-mode applications in addition to the native VxWorks APIs.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous I/O</td>
<td>aioPxLib</td>
</tr>
<tr>
<td>Buffer manipulation</td>
<td>bLib</td>
</tr>
<tr>
<td>Clock facility</td>
<td>clockLib</td>
</tr>
<tr>
<td>Directory handling</td>
<td>dirLib</td>
</tr>
<tr>
<td>Environment handling</td>
<td>C Library</td>
</tr>
<tr>
<td>Environment information</td>
<td>sysconf and uname</td>
</tr>
<tr>
<td>File duplication</td>
<td>ioLib</td>
</tr>
<tr>
<td>File management</td>
<td>fsPxLib and ioLib</td>
</tr>
<tr>
<td>I/O functions</td>
<td>ioLib</td>
</tr>
<tr>
<td>Options handling</td>
<td>getopt</td>
</tr>
<tr>
<td>POSIX message queues</td>
<td>mqPxLib</td>
</tr>
<tr>
<td>POSIX semaphores</td>
<td>semPxLib</td>
</tr>
<tr>
<td>POSIX timers</td>
<td>timerLib</td>
</tr>
<tr>
<td>POSIX threads</td>
<td>pthreadLib</td>
</tr>
<tr>
<td>Standard I/O and some ANSI</td>
<td>C Library</td>
</tr>
<tr>
<td>Math</td>
<td>C Library</td>
</tr>
<tr>
<td>Memory allocation</td>
<td>memLib</td>
</tr>
<tr>
<td>Network/Socket APIs</td>
<td>network libraries</td>
</tr>
<tr>
<td>String manipulation</td>
<td>C Library</td>
</tr>
<tr>
<td>Trace facility</td>
<td>pxTraceLib</td>
</tr>
<tr>
<td>Wide character support</td>
<td>C library</td>
</tr>
</tbody>
</table>

Table 4-2  POSIX Libraries
Checking for POSIX Support at Run-time

A POSIX application can use the following APIs at run-time to determine the status of POSIX support in the system:

- The `sysconf()` routine returns the current values of the configurable system variables, allowing an application to determine whether an optional feature is supported or not, and the precise value of system's limits.

- The `confstr()` routine returns a string associated with a system variable. With this release, the `confstr()` routine returns a string only for the system’s default path.

- The `uname()` routine lets an application get information about the system on which it is running. The identification information provided for VxWorks is the system name (VxWorks), the network name of the system, the system’s release number, the machine name (BSP model), the architecture’s endianness, the kernel version number, the processor name (CPU family), the BSP revision level, and the system’s build date.

4.4 Standard C Library: Libc

The standard C library for user space (RTP applications) complies with the POSIX PSE52 profile. For information about the APIs provided in the C library, see the VxWorks Application API Reference and the Dinkum C Library Reference Manual.
4.5 POSIX Header Files

The POSIX 1003.1 standard defines a set of header files as part of the application development environment. The VxWorks user-side development environment provides more POSIX header files than the kernel’s, and their content is also more in agreement with the POSIX standard than the kernel header files.

The POSIX header files available for the user development environment are listed in Table 4-3. The PSE52 column indicates compliance with the POSIX PSE52 profile.

<table>
<thead>
<tr>
<th>Header File</th>
<th>PSE52</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aio.h</td>
<td></td>
<td>asynchronous input and output</td>
</tr>
<tr>
<td>assert.h</td>
<td></td>
<td>verify program assertion</td>
</tr>
<tr>
<td>complex.h</td>
<td></td>
<td>complex arithmetic</td>
</tr>
<tr>
<td>ctype.h</td>
<td></td>
<td>character types</td>
</tr>
<tr>
<td>dirent.h</td>
<td></td>
<td>format of directory entries</td>
</tr>
<tr>
<td>dlfcn.h</td>
<td></td>
<td>dynamic linking</td>
</tr>
<tr>
<td>errno.h</td>
<td></td>
<td>system error numbers</td>
</tr>
<tr>
<td>fcntl.h</td>
<td></td>
<td>file control options</td>
</tr>
<tr>
<td>fenv.h</td>
<td></td>
<td>floating-point environment</td>
</tr>
<tr>
<td>float.h</td>
<td></td>
<td>floating types</td>
</tr>
<tr>
<td>inttypes.h</td>
<td></td>
<td>fixed size integer types</td>
</tr>
<tr>
<td>iso646.h</td>
<td></td>
<td>alternative spellings</td>
</tr>
<tr>
<td>limits.h</td>
<td></td>
<td>implementation-defined constants</td>
</tr>
<tr>
<td>locale.h</td>
<td></td>
<td>category macros</td>
</tr>
<tr>
<td>math.h</td>
<td></td>
<td>mathematical declarations</td>
</tr>
<tr>
<td>mqueue.h</td>
<td></td>
<td>message queues</td>
</tr>
<tr>
<td>pthread.h</td>
<td></td>
<td>pthreads</td>
</tr>
</tbody>
</table>
### POSIX Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched.h</td>
<td>execution scheduling</td>
</tr>
<tr>
<td>search.h</td>
<td>search tables</td>
</tr>
<tr>
<td>semaphore.h</td>
<td>semaphores</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>stack environment declarations</td>
</tr>
<tr>
<td>signal.h</td>
<td>signals</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>boolean type and values</td>
</tr>
<tr>
<td>stddef.h</td>
<td>standard type definitions</td>
</tr>
<tr>
<td>stdint.h</td>
<td>integer types</td>
</tr>
<tr>
<td>stdio.h</td>
<td>standard buffered input/output</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>standard library definitions</td>
</tr>
<tr>
<td>string.h</td>
<td>string operations</td>
</tr>
<tr>
<td>strings.h</td>
<td>string operations</td>
</tr>
<tr>
<td>sys/mman.h</td>
<td>memory management declarations</td>
</tr>
<tr>
<td>sys/resource.h</td>
<td>definitions for XSI resource operations</td>
</tr>
<tr>
<td>sys/select.h</td>
<td>select types</td>
</tr>
<tr>
<td>sys/stat.h</td>
<td>data returned by the <code>stat()</code> function</td>
</tr>
<tr>
<td>sys/types.h</td>
<td>data types</td>
</tr>
<tr>
<td>sys/un.h</td>
<td>definitions for UNIX domain sockets</td>
</tr>
<tr>
<td>sys/utsname.h</td>
<td>system name structure</td>
</tr>
<tr>
<td>sys/wait.h</td>
<td>declarations for waiting</td>
</tr>
<tr>
<td>tgmath.h</td>
<td>type-generic macros</td>
</tr>
<tr>
<td>time.h</td>
<td>time types</td>
</tr>
<tr>
<td>trace.h</td>
<td>trace facility</td>
</tr>
</tbody>
</table>
POSIX Namespace

The POSIX.1 standard guarantees that the namespace of strictly conforming POSIX applications is not polluted by POSIX implementation symbols.

For VxWorks, this namespace isolation can be enforced for all symbols that can be included by way of POSIX PSE52 header files, but does not extend to VxWorks library symbols (for more information, see Limitations to Namespace Isolation, p.174).

POSIX namespace isolation can, of course, only be enforced for applications that only include header files that are defined by the POSIX.1 standard. It cannot be enforced for applications that include native VxWorks header files.

Limitations to Namespace Isolation

For VxWorks, POSIX features are implemented using native VxWorks features. When VxWorks system libraries are linked with POSIX applications, therefore, some native VxWorks public functions and public variable identifiers become part of the application's namespace. Since native VxWorks public functions and public variable identifiers do not abide by POSIX naming conventions (in particular they do not generally start with an underscore) they may pollute a POSIX application's own namespace. In other words, the identifiers used by VxWorks native public functions and public variables must be considered as reserved even by POSIX applications.

Table 4-3  POSIX Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>PSE52 Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unistd.h</td>
<td>standard symbolic constants and types</td>
</tr>
<tr>
<td>utime.h</td>
<td>access and modification times structure</td>
</tr>
<tr>
<td>wchar.h</td>
<td>wide-character handling</td>
</tr>
<tr>
<td>wctype.h</td>
<td>wide-character classification and mapping utilities</td>
</tr>
</tbody>
</table>
POSIX Header Files

The POSIX header files that have been implemented in compliance with the POSIX PSE52 profile are identified in Table 4-3.

These header files are also used by traditional VxWorks applications, and may be included in native VxWorks header files (that is, header files that are not described by standards like POSIX or ANSI).

Using POSIX Feature-Test Macros

POSIX provides feature-test macros used during the compilation process to ensure application namespace isolation. These macros direct the compiler’s pre-processor to exclude all symbols that are not part of the POSIX.1 authorized symbols in the POSIX header files.

The supported POSIX feature test macros and their values are as follows:

- 
  -POSIX_C_SOURCE set to 200112L
- 
  _XOPEN_SOURCE set to 600
- 
  _POSIX_AEP_RT_CONTROLLER_C_SOURCE set to 200312L

Wind River supports use of these macros for use with POSIX PSE52 header files (see 4.5 POSIX Header Files, p.172).

The macros must be defined before any other header file inclusions. Defining any of the three feature test macros with the appropriate value is sufficient to enforce POSIX namespace isolation with regard to header file symbols.

Note that the macros must be used with the values provided here to ensure the appropriate results. Also note that the POSIX.1 and POSIX.13 standards could evolve so that using different supported values for those macros, or defining these macros in various combinations, might yield different results with regard to which symbols are made visible to the application.

The _POSIX_SOURCE feature test macro is not supported since it has been superseded by _POSIX_C_SOURCE.

⚠️ CAUTION: Only POSIX header files may be included in a POSIX application that is compiled with the POSIX feature test macros. Inclusion of any other header files may pollute the application’s namespace, and may also result in compilation failures.
4.7 POSIX Process Support

VxWorks provides support for a user-mode process model with real-time processes (RTPs). Table 4-4 provides a partial list of the POSIX APIs provided for manipulating processes. They take a `_pid_` argument (also known as an RTP_ID in VxWorks).

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atexit()</code></td>
<td>Register a handler to be called at <code>exit()</code>.</td>
</tr>
<tr>
<td><code>_exit()</code></td>
<td>Terminate the calling process (system call).</td>
</tr>
<tr>
<td><code>exit()</code></td>
<td>Terminate a process, calling <code>atexit()</code> handlers.</td>
</tr>
<tr>
<td><code>getpid()</code></td>
<td>Get the process ID of the current process.</td>
</tr>
<tr>
<td><code>getppid()</code></td>
<td>Get the process ID of the parent’s process ID.</td>
</tr>
<tr>
<td><code>kill()</code></td>
<td>Send a signal to a process.</td>
</tr>
<tr>
<td><code>raise()</code></td>
<td>Send a signal to the caller's process.</td>
</tr>
<tr>
<td><code>wait()</code></td>
<td>Wait for any child process to die.</td>
</tr>
<tr>
<td><code>waitpid()</code></td>
<td>Wait for a specific child process to die.</td>
</tr>
</tbody>
</table>

Basic VxWorks process facilities, including these routines, are provided with the INCLUDE_RTP component.

⚠️ CAUTION: If a PSE52-conforming application relies on C99 constructs, the Wind River Compiler must be used. The version of the GNU compiler provided with this release does not support the C99 language constructs.
4.8 POSIX Clocks and Timers

VxWorks provides POSIX.1 standard clock and timer interfaces.

**POSIX Clocks**

POSIX defines various software (virtual) clocks, which are identified as the `CLOCK_REALTIME` clock, `CLOCK_MONOTONIC` clock, process CPU-time clocks, and thread CPU-time clocks. These clocks all use one system hardware timer.

The real-time clock and the monotonic clock are system-wide clocks, and are therefore supported for both the VxWorks kernel and processes. The process CPU-time clocks are not supported in VxWorks. The thread CPU-time clocks are supported for POSIX threads running in processes. A POSIX thread can use the real-time clock, the monotonic clock, and a thread CPU-time clock for its application.

The real-time clock can be reset (but only from the kernel). The monotonic clock cannot be reset, and provides the time that has elapsed since the system booted.

The real-time clock can be accessed with the POSIX clock and timer routines by using the `clock_id` parameter `CLOCK_REALTIME`. A real-time clock can be reset at run time with a call to `clock_settime()` from within the kernel (not from a process).

The monotonic clock can be accessed by calling `clock_gettime()` with a `clock_id` parameter of `CLOCK_MONOTONIC`. A monotonic clock keeps track of the time that has elapsed since system startup; that is, the value returned by `clock_gettime()` is the amount of time (in seconds and nanoseconds) that has passed since the system booted. A monotonic clock cannot be reset. Applications can therefore rely on the fact that any measurement of a time interval that they might make has not been falsified by a call to `clock_settime()`.

Both `CLOCK_REALTIME` and `CLOCK_MONOTONIC` are defined in `time.h`.

In addition to system wide clocks, VxWorks supports thread CPU-time clocks for individual POSIX threads. A thread CPU-time clock measures the execution time of a specific pthread, including the time that the system spends executing system services on behalf of the pthread (that is, the time it runs in both user and kernel mode). The initial execution time is set to zero when the pthread is created.

Thread CPU-time clocks are implemented only for pthreads that run in processes (and not in the kernel). These clocks are accessible only through POSIX APIs.

Thread CPU-time clocks provide a means for detecting and managing situations in which a pthread overruns its assigned maximum execution time. Bounding the
execution times of the pthreads in an application increases predictability and reliability.

Each POSIX thread has its own CPU-time clock to measure its execution time of pthread, which can be identified by using the `pthread_getcpuclockid()` routine. The `CLOCK_THREAD_CPUTIME_ID clock_id` parameter refers to the calling thread's CPU-time clock.

See Table 4-5 for a list of the POSIX clock routines. The obsolete VxWorks-specific POSIX extension `clock_setres()` is provided for backwards-compatibility purposes. For more information about clock routines, see the API reference for `clockLib`.

### Table 4-5  POSIX Clock Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clock_getres()</code></td>
<td>Get the clock resolution (CLOCK_REALTIME, CLOCK_MONOTONIC, and thread CPU-time).</td>
</tr>
<tr>
<td><code>clock_setres()</code></td>
<td>Set the clock resolution. Obsolete VxWorks-specific POSIX extension.</td>
</tr>
<tr>
<td><code>clock_gettime()</code></td>
<td>Get the current clock time (CLOCK_REALTIME, CLOCK_MONOTONIC, and thread CPU-time).</td>
</tr>
<tr>
<td><code>clock_settime()</code></td>
<td>Set the thread CPU-time clock (fails for CLOCK_REALTIME or CLOCK_MONOTONIC).</td>
</tr>
<tr>
<td><code>clock_nanosleep()</code></td>
<td>Suspend the current pthread (or task) for a specified amount of time.</td>
</tr>
<tr>
<td><code>pthread_getcpuclockid()</code></td>
<td>Get the clock ID for the CPU-time clock (for use with <code>clock_gettime()</code> and <code>clock_settime()</code>).</td>
</tr>
</tbody>
</table>

To include the `clockLib` library in the system, configure VxWorks with the `INCLUDE_POSIX_CLOCKS` component. For thread CPU-time clocks, the `INCLUDE_POSIX_PTHREAD_SCHEDULER` and `INCLUDE_POSIX_THREAD_CPUTIME` components must be used as well.

Process-based (RTP) applications are automatically linked with the `clockLib` library when they are compiled. The library is automatically initialized when the process starts.
POSIX Timers

The POSIX timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to create, set, and delete a timer.

Timers are created based on clocks. In the kernel, the CLOCK_REALTIME and CLOCK_MONOTONIC clocks are supported for timers. In processes, the CLOCK_REALTIME clock, CLOCK_MONOTONIC clock, and thread CPU-time clocks (including CLOCK_THREAD_CPUTIME_ID clock) are supported.

When a timer goes off, the default signal, SIGALRM, is sent to the task. To install a signal handler that executes when the timer expires, use the sigaction() routine (see 3.3.10 Signals, p.149).

See Table 4-6 for a list of the POSIX timer routines. The VxWorks timerLib library includes a set of VxWorks-specific POSIX extensions: timer_open(), timer_close(), timer_cancel(), timer_connect(), and timer_unlink(). These routines allow for an easier and more powerful use of POSIX timers on VxWorks. For more information, see the VxWorks API reference for timerLib.

Table 4-6  POSIX Timer Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer_create()</td>
<td>Allocate a timer using the specified clock for a timing base</td>
</tr>
<tr>
<td></td>
<td>(CLOCK_REALTIME, CLOCK_MONOTONIC, or thread</td>
</tr>
<tr>
<td></td>
<td>CPU-time of the calling thread).</td>
</tr>
<tr>
<td>timer_delete()</td>
<td>Remove a previously created timer.</td>
</tr>
<tr>
<td>timer_open()</td>
<td>Open a name timer.</td>
</tr>
<tr>
<td></td>
<td>VxWorks-specific POSIX extension.</td>
</tr>
<tr>
<td>timer_close()</td>
<td>Close a name timer.</td>
</tr>
<tr>
<td></td>
<td>VxWorks-specific POSIX extension.</td>
</tr>
<tr>
<td>timer_gettime()</td>
<td>Get the remaining time before expiration and the reload</td>
</tr>
<tr>
<td></td>
<td>value.</td>
</tr>
<tr>
<td>timer_getoverrun()</td>
<td>Return the timer expiration overrun.</td>
</tr>
<tr>
<td>timer_settime()</td>
<td>Set the time until the next expiration and arm timer.</td>
</tr>
<tr>
<td>timer_cancel()</td>
<td>Cancel a timer.</td>
</tr>
<tr>
<td></td>
<td>VxWorks-specific POSIX extension.</td>
</tr>
</tbody>
</table>
Example 4-1 POSIX Timers

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

/* includes */
#include <vxWorks.h>
#include <time.h>

int createTimer (void)
{
    timer_t timerid;
    /* create timer */
    if (timer_create (CLOCK_REALTIME, NULL, &timerid) == ERROR)
    {
        printf ("create FAILED\n");
        return (ERROR);
    }
    return (OK);
}

The POSIX nanosleep() routine provides specification of sleep or delay time in units of seconds and nanoseconds, in contrast to the ticks used by the VxWorks taskDelay() function. Nevertheless, the precision of both is the same, and is determined by the system clock rate; only the units differ.

To include the timerLib library in a system, configure VxWorks with the INCLUDE_POSIX_TIMERS component.

Process-based (RTP) applications are automatically linked with the timerLib library when they are compiled. The library is automatically initialized when the process starts.

Table 4-6 POSIX Timer Routines (cont'd)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer_connect()</td>
<td>Connect a user routine to the timer signal.</td>
</tr>
<tr>
<td></td>
<td>VxWorks-specific POSIX extension.</td>
</tr>
<tr>
<td>timer_unlink()</td>
<td>Unlink a named timer.</td>
</tr>
<tr>
<td></td>
<td>VxWorks-specific POSIX extension.</td>
</tr>
<tr>
<td>nanosleep()</td>
<td>Suspend the current pthread (task) until the time interval elapses.</td>
</tr>
<tr>
<td>sleep()</td>
<td>Delay for a specified amount of time.</td>
</tr>
<tr>
<td>alarm()</td>
<td>Set an alarm clock for delivery of a signal.</td>
</tr>
</tbody>
</table>

VxWorks
Application Programmer's Guide, 6.4
4.9 POSIX Asynchronous I/O

POSIX asynchronous I/O (AIO) routines are provided by the `aioPxLib` library. The VxWorks AIO implementation meets the specification of the POSIX 1003.1 standard. For more information, see 6.6 Asynchronous Input/Output, p.277.

4.10 POSIX Advisory File Locking

POSIX advisory file locking provides byte-range locks on POSIX-conforming files (for VxWorks, this means files in an HRFS file system). The VxWorks implementation meets the specification of the POSIX 1003.1 standard.

POSIX advisory file locking is provided through the `fcntl()` file control function. To include POSIX advisory file locking facilities in VxWorks, configure the system with the `INCLUDE_POSIX_ADVISORY_FILE_LOCKING` component.

The VxWorks implementation of advisory file locking involves a behavioral difference with regard to deadlock detection because VxWorks processes are not scheduled. Note that this distinction only matters if you have multiple pthreads (or tasks) within one process (RTP).

According to POSIX, advisory locks are identified by a process ID, and when a process exits all of its advisory locks are destroyed, which is true for VxWorks. But because VxWorks processes cannot themselves be scheduled, individual advisory locks on a given byte range of a file have two owners: the pthread (or task) that actually holds the lock, and the process that contains the pthread. In addition, the calculation of whether one lock would deadlock another lock is done on a pthread basis, rather than a process basis.

This means that deadlocks are detected if the pthread requesting a new lock would block on any pthread (in any given process) that is currently blocked (whether directly or indirectly) on any advisory lock that is held by the requesting pthread. Immediate-blocking detection (`F_SETLK` requests) always fail immediately if the requested byte range cannot be locked without waiting for some other lock, regardless of the identity of the owner of that lock.
4.11 POSIX Page-Locking Interface

The real-time extensions of the POSIX 1003.1 standard are used with operating systems that perform paging and swapping. On such systems, applications that attempt real-time performance can use the POSIX page-locking facilities to protect certain blocks of memory from paging and swapping.

VxWorks does not support memory paging and swapping because the serious delays in execution time that they cause are undesirable in a real-time system. However, page-locking routines can be included in VxWorks to facilitate porting POSIX applications to VxWorks.

These routines do not perform any function except to validate the parameters passed, and if the addresses are passed, they are validated to ensure they are resident in memory. The routines do not perform any other function, as all pages are always kept in memory.

The POSIX page-locking routines are part of the memory management library, mmanPxLib, and are listed in Table 4-7.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Purpose on Systems with Paging or Swapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlockall()</td>
<td>Locks into memory all pages used by a task.</td>
</tr>
<tr>
<td>munlockall()</td>
<td>Unlocks all pages used by a task.</td>
</tr>
<tr>
<td>mlock()</td>
<td>Locks a specified page.</td>
</tr>
<tr>
<td>munlock()</td>
<td>Unlocks a specified page.</td>
</tr>
</tbody>
</table>

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

Process-based (RTP) applications are automatically linked with the mmanPxLib library when they are compiled.
4.12 POSIX Threads

POSIX threads (also known as pthreads) are similar to VxWorks tasks, but with additional characteristics. In VxWorks pthreads are implemented on top of native tasks, but maintain pthread IDs that differ from the IDs of the underlying tasks.

The main reasons for including POSIX thread support in VxWorks are the following:

- For porting POSIX applications to VxWorks.
- To make use of the POSIX thread scheduler in real-time processes (including concurrent scheduling policies).

For information about POSIX thread scheduler, see 4.14 POSIX and VxWorks Scheduling, p.197.

4.12.1 POSIX Thread Stack Guard Zones

Execution-stack guard zones can be used with POSIX threads. A SIGSEGV signal is then generated when a stack overflow or underflow occurs.

Execution stack overflow protection can be provided for individual pthreads with a call to the `pthread_attr_setguardsize()` routine before the pthread is created. By default the size of a pthread guard zone is one architecture-specific page (`VM_PAGESIZE`). It can be set to any value with the `pthread_attr_setguardsize()` routine, but the actual size is always rounded to a multiple of a number of memory pages.

The size of the guard area must be set in a pthread attribute before a thread is created that uses the attribute. Once a thread is created, the size of its stack guard cannot be changed. If the size attribute is changed after the thread is created, it does not affect the thread that has already been created, but it does apply to the next thread created that uses the attribute. If a pthread’s stack is created with guard protection, the `pthread_attr_getguardsize()` routine can be used to get the size of the guard zone.

Note that the default overflow guard zone size for pthreads is independent of the size of the execution stack overflow guard zone for tasks (with is defined globally for tasks, and provided for them with the INCLUDE_RTP component).

Also note that pthreads are provided with execution stack underflow guard zones by default (they are provided for both pthreads and tasks by the INCLUDE_RTP component).
For information about task guard zones, see *Task Stack*, p. 89.

### 4.12.2 POSIX Thread Attributes

A major difference between VxWorks tasks and POSIX threads is the way in which options and settings are specified. For VxWorks tasks these options are set with the task creation API, usually `taskSpawn()`.

POSIX threads, on the other hand, have characteristics that are called attributes. Each attribute contains a set of values, and a set of access routines to retrieve and set those values. You specify all pthread attributes before pthread creation in the attributes object `pthread_attr_t`. In a few cases, you can dynamically modify the attribute values of a pthread after its creation.

### 4.12.3 VxWorks-Specific Pthread Attributes

The VxWorks implementation of POSIX threads provides two additional pthread attributes (which are POSIX extensions)—pthread name and pthread options—as well as routines for accessing them.

#### Pthread Name

While POSIX threads are not named entities, the VxWorks tasks upon which they are based are named. By default the underlying task elements are named `pthrNumber` (for example, `pthr3`). The number part of the name is incremented each time a new thread is created (with a roll-over at $2^{32} - 1$). It is, however, possible to name these tasks using the thread name attribute.

- **Attribute Name:** `threadname`
- **Possible Values:** a null-terminated string of characters
- **Default Value:** none (the default naming policy is used)
- **Access Functions (VxWorks-specific POSIX extensions):** `pthread_attr_setname()` and `pthread_attr_getname()`

#### Pthread Options

POSIX threads are agnostic with regard to target architecture. Some VxWorks tasks, on the other hand, may be created with specific options in order to benefit from certain features of the architecture. For example, for the Altivec-capable PowerPC architecture, tasks must be created with the `VX_ALTIVEC_TASK` in order...
to make use of the Altivec processor. The pthread options attribute can be used to set such options for the VxWorks task upon which the POSIX thread is based.

- Attribute Name: **threadoptions**
- Possible Values: the same as the VxWorks task options. See **taskLib.h**
- Default Value: none (the default task options are used)
- Access Functions (VxWorks-specific POSIX extensions): `pthread_attr_setopt()` and `pthread_attr_getopt()`

### 4.12.4 Specifying Attributes when Creating Pthreads

The following examples create a pthread using the default attributes and use explicit attributes.

**Example 4-2  Creating a pthread Using Explicit Scheduling Attributes**

```c
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 4-3  Creating a pthread Using Default Attributes**

```c
pthread_t tid;
int ret;
/* create the pthread with NULL attributes to designate default values */
ret = pthread_create(&tid, NULL, entryFunction, entryArg);
```
Example 4-4  Designating Your Own Stack for a pthread

```c
pthread_t threadId;
pthread_attr_t attr;
void * stackaddr = NULL;
int stacksize = 0;

/* initialize the thread's attributes */
pthread_attr_init (&attr);

/*
 * Allocate memory for a stack region for the thread. Malloc() is used
 * for simplification since a real-life case is likely to use
 * memPartAlloc()
 * on the kernel side, or mmap() on the user side.
 */
stacksize = 2 * 4096 /* let's allocate two pages */
stackaddr = malloc (stacksize);
if (stackbase == NULL)
{
    printf ("FAILED: mystack: malloc failed\n");
    return (-1);
}

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

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

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

/* create the pthread */
ret = pthread_create (&threadId, &attr, mystack_thread, 0);
```
4.12.5 POSIX Thread Creation and Management

VxWorks provides many POSIX thread routines. Table 4-8 lists a few that are directly relevant to pthread creation or execution. See the VxWorks API reference for information about the other routines, and more details about all of them.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create( )</td>
<td>Create a pthread.</td>
</tr>
<tr>
<td>pthread_cancel( )</td>
<td>Cancel the execution of a pthread</td>
</tr>
<tr>
<td>pthread_detach( )</td>
<td>Detach a running pthread so that it cannot be joined by another pthread.</td>
</tr>
<tr>
<td>pthread_join( )</td>
<td>Wait for a pthread to terminate.</td>
</tr>
<tr>
<td>pthread_getschedparam( )</td>
<td>Dynamically set value of scheduling priority attribute.</td>
</tr>
<tr>
<td>pthread_setschedparam( )</td>
<td>Dynamically set scheduling priority and policy parameter.</td>
</tr>
<tr>
<td>pthread_setschedprio( )</td>
<td>Dynamically set scheduling priority parameter.</td>
</tr>
<tr>
<td>sched_get_priority_max( )</td>
<td>Get the maximum priority that a pthread can get.</td>
</tr>
<tr>
<td>sched_get_priority_min( )</td>
<td>Get the minimum priority that a pthread can get.</td>
</tr>
<tr>
<td>sched_rr_get_interval( )</td>
<td>Get the time quantum of execution of the round-robin policy.</td>
</tr>
<tr>
<td>sched_yield( )</td>
<td>Relinquishes the CPU.</td>
</tr>
<tr>
<td>pthread_getconcurrency( )</td>
<td>Get level of concurrency.</td>
</tr>
<tr>
<td></td>
<td>In VxWorks this routine does nothing.</td>
</tr>
<tr>
<td>pthread_setconcurrency( )</td>
<td>Set level of concurrency</td>
</tr>
<tr>
<td></td>
<td>In VxWorks this routine does nothing.</td>
</tr>
</tbody>
</table>
4.12.6 POSIX Thread Attribute Access

The POSIX attribute-access routines are described in Table 4-9. The VxWorks-specific POSIX extension routines are described in section 4.12.3 VxWorks-Specific Pthread Attributes, p.184.

Table 4-9 POSIX Thread Attribute-Access Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_getstacksize()</td>
<td>Get value of the stack size attribute.</td>
</tr>
<tr>
<td>pthread_attr_setstacksize()</td>
<td>Set the stack size attribute.</td>
</tr>
<tr>
<td>pthread_attr_getstackaddr()</td>
<td>Get value of stack address attribute.</td>
</tr>
<tr>
<td>pthread_attr_setstackaddr()</td>
<td>Set value of stack address attribute.</td>
</tr>
<tr>
<td>pthread_attr_getstack()</td>
<td>Get the values of the stack address and size attributes.</td>
</tr>
<tr>
<td>pthread_attr_setstack()</td>
<td>Set the values of the stack address and size attributes.</td>
</tr>
<tr>
<td>pthread_attr_getguardsize()</td>
<td>Get the stack guard zone size attribute.</td>
</tr>
<tr>
<td>pthread_attr_setguardsize()</td>
<td>Set the stack guard zone size attribute.</td>
</tr>
<tr>
<td>pthread_attr_getdetachstate()</td>
<td>Get value of detachstate attribute (joinable or detached).</td>
</tr>
<tr>
<td>pthread_attr_setdetachstate()</td>
<td>Set value of detachstate attribute (joinable or detached).</td>
</tr>
<tr>
<td>pthread_attr_getscope()</td>
<td>Get contention scope.</td>
</tr>
<tr>
<td>pthread_attr_setscope()</td>
<td>Set contention scope.</td>
</tr>
<tr>
<td>pthread_attr_getinheritsched()</td>
<td>Get value of scheduling-inheritance attribute.</td>
</tr>
<tr>
<td>pthread_attr_setinheritsched()</td>
<td>Set value of scheduling-inheritance attribute.</td>
</tr>
</tbody>
</table>

Only PTHREAD_SCOPE_SYSTEM is supported for VxWorks.
4.12 POSIX Threads

4.12.7 POSIX Thread Private Data

POSIX threads can store and access private data; that is, pthread-specific data. They use a key maintained for each pthread by the pthread library to access that data. A key corresponds to a location associated with the data. It is created by calling `pthread_key_create()` and released by calling `pthread_key_delete()`. The location is accessed by calling `pthread_getspecific()` and `pthread_setspecific()`. This location represents a pointer to the data, and not the data itself, so there is no limitation on the size and content of the data associated with a key.

The pthread library supports a maximum of 256 keys for all the pthreads in a process.

The `pthread_key_create()` routine has an option for a destructor function, which is called when the creating pthread exits or is cancelled, if the value associated with the key is non-NULL.

This destructor function frees the storage associated with the data itself, and not with the key. It is important to set a destructor function for preventing memory leaks.
leaks to occur when the pthread that allocated memory for the data is cancelled. The key itself should be freed as well, by calling `pthread_key_delete()`, otherwise the key cannot be reused by the pthread library.

### 4.12.8 POSIX Thread Cancellation

POSIX provides a mechanism, called *cancellation*, to terminate a pthread gracefully. There are two types of cancellation: *deferred* and *asynchronous*.

Deferred cancellation causes the pthread to explicitly check to see if it was cancelled. This happens in one of the two following ways:

- The code of the pthread executes calls to `pthread_testcancel()` at regular intervals.
- The pthread calls a function that contains a *cancellation point* during which the pthread may be automatically cancelled.

Asynchronous cancellation causes the execution of the pthread to be forcefully interrupted and a handler to be called, much like a signal.\(^1\)

Automatic cancellation points are library routines that can block the execution of the pthread for a lengthy period of time.

**NOTE:** While the `msync()`, `fcntl()`, and `tcdrain()` routines are mandated POSIX 1003.1 cancellation points, they are not provided with VxWorks for this release.

The POSIX cancellation points provided in VxWorks libraries are described in Table 4-10.

<table>
<thead>
<tr>
<th>Library</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>aioPxLib</td>
<td><code>aio_suspend()</code></td>
</tr>
<tr>
<td>ioLib</td>
<td><code>creat()</code>, <code>open()</code>, <code>read()</code>, <code>write()</code>, <code>close()</code>, <code>fsync()</code>, <code>fdatasync()</code></td>
</tr>
<tr>
<td>mqPxLib</td>
<td><code>mq_receive()</code>, <code>mq_send()</code></td>
</tr>
</tbody>
</table>

---

\(^1\) Asynchronous cancellation is actually implemented with a special signal, `SIGCNCL`, which users should be careful not to block or to ignore.
Table 4-10  Pthread Cancellation Points in VxWorks Libraries  (cont'd)

<table>
<thead>
<tr>
<th>Library</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthreadLib</td>
<td>pthread_cond_timedwait(), pthread_cond_wait(), pthread_join(), pthread_testcancel()</td>
</tr>
<tr>
<td>semPxLib</td>
<td>sem_wait()</td>
</tr>
<tr>
<td>sigLib</td>
<td>pause(), sigsuspend(), sigtimedwait(), sigwait(), sigwaitinfo(), waitpid()</td>
</tr>
<tr>
<td>timerLib</td>
<td>sleep(), nanosleep()</td>
</tr>
</tbody>
</table>

Routines that can be used with cancellation points of pthreads are listed in Table 4-11.

Table 4-11  Pthread Cancellation Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cancel()</td>
<td>Cancel execution of a pthread.</td>
</tr>
<tr>
<td>pthread_testcancel()</td>
<td>Create a cancellation point in the calling pthread.</td>
</tr>
<tr>
<td>pthread_setcancelstate()</td>
<td>Enables or disables cancellation.</td>
</tr>
<tr>
<td>pthread_setcanceltype()</td>
<td>Selects deferred or asynchronous cancellation.</td>
</tr>
<tr>
<td>pthread_cleanup_push()</td>
<td>Registers a function to be called when the pthread is cancelled, exits, or calls pthread_cleanup_pop() with a non-null run parameter.</td>
</tr>
<tr>
<td>pthread_cleanup_pop()</td>
<td>Unregisters a function previously registered with pthread_cleanup_push(). This function is immediately executed if the run parameter is non-null.</td>
</tr>
</tbody>
</table>
4.13 POSIX Thread Mutexes and Condition Variables

Pthread mutexes (mutual exclusion variables) and condition variables comply with the POSIX.1-2001 standard. Like POSIX threads, mutexes and condition variables have attributes associated with them. Mutex attributes are held in a data type called `pthread_mutexattr_t`, which contains three attributes: priority ceiling, protocol and type.

The routines used to manage these attributes are described below. For more information about these and other mutex routines, see the VxWorks API reference for pthreadLib.

4.13.1 Thread Mutexes

The routines that can be used to act directly on a mutex object and on the mutex attribute object are listed in Table 4-12 and Table 4-13 (respectively).

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_mutex_destroy()</code></td>
<td>Destroy a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_init()</code></td>
<td>Initialize a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_getprioceiling()</code></td>
<td>Get the priority ceiling of a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_setprioceiling()</code></td>
<td>Set the priority ceiling of a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_lock()</code></td>
<td>Lock a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_trylock()</code></td>
<td>Check and lock a mutex if available.</td>
</tr>
<tr>
<td><code>pthread_mutex_unlock()</code></td>
<td>Unlock a mutex.</td>
</tr>
<tr>
<td><code>pthread_mutex_timedlock()</code></td>
<td>Lock a mutex with timeout.</td>
</tr>
</tbody>
</table>
The `type` mutex attribute controls the behavior of the mutex object when a pthread attempts to lock and unlock a mutex. The possible values are `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_ERRORCHECK`, `PTHREAD_MUTEX_RECURSIVE`, and `PTHREAD_MUTEX_DEFAULT` which is mapped to the `PTHREAD_MUTEX_NORMAL` type on VxWorks.

Mutexes of types `PTHREAD_MUTEX_NORMAL` and `PTHREAD_MUTEX_DEFAULT` do not detect deadlocks, so a pthread will deadlock if it attempts to lock a mutex it has locked already. A pthread attempting to unlock a mutex of this type that is locked by another pthread, or is already unlocked, will get the `EPERM` error.

Mutexes of type `PTHREAD_MUTEX_ERRORCHECK` do detect a deadlock situation, so a pthread will get the `EDEADLK` error if it attempts to lock a mutex that it has locked already. A pthread attempting to unlock a mutex of this type that is locked by another pthread, or is already unlocked, will get the `EPERM` error.

Mutexes of type `PTHREAD_MUTEX_RECURSIVE` allow a pthread to relock a mutex that it has already locked. The same number of unlocks as locks is required to

Table 4-13  POSIX Routines Acting on a Mutex Attribute Object

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutexattr_init()</td>
<td>Initialize mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_destroy()</td>
<td>Destroy mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_getprioceiling()</td>
<td>Get <code>prioceiling</code> attribute of mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_setprioceiling()</td>
<td>Set <code>prioceiling</code> attribute of mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_getprotocol()</td>
<td>Get <code>protocol</code> attribute of mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_setprotocol()</td>
<td>Set <code>protocol</code> attribute of mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_gettype()</td>
<td>Get a mutex <code>type</code> attribute.</td>
</tr>
<tr>
<td>pthread_mutexattr_settype()</td>
<td>Set a mutex <code>type</code> attribute.</td>
</tr>
</tbody>
</table>
release the mutex. A pthread attempting to unlock a mutex of this type that has
been locked by another pthread, or is already unlocked, will get the EPERM error.

Protocol Mutex Attribute

The protocol mutex attribute defines how the mutex variable deals with the
priority inversion problem (which is described in the section for VxWorks
mutual-exclusion semaphores; see Mutual-Exclusion Semaphores, p.113).

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

By default, a mutex is created with the PTHREAD_PRIO_NONE protocol, which
ensures that owning the mutex does not modify the priority and scheduling
characteristics of a pthread. This may, however, not be appropriate in situations in
which a low priority pthread can lock a mutex that is also required by higher
priority threads. The PTHREAD_PRIO_INHERIT and PTHREAD_PRIO_PROTECT
protcols can be used to address this issue.

The PTHREAD_PRIO_INHERIT value is used to create a mutex with priority
inheritance—and is equivalent to the association of SEM_Q_PRIORITY and
SEM_INVERSION_SAFE options used with semMCreate(). A pthread owning a
mutex variable created with the PTHREAD_PRIO_INHERIT value inherits the
priority of any higher-priority pthread waiting for the mutex and executes at this
elevated priority until it releases the mutex, at which points it returns to its original
priority.

Because it might not be desirable to elevate a lower-priority pthread to a priority
above a certain level, POSIX defines the notion of priority ceiling, described below.
Mutual-exclusion variables created with priority protection use the
PTHREAD_PRIO_PROTECT value.
Priority Ceiling Mutex Attribute

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

- **Attribute Name**: **prioceiling**
- **Possible Values**: any valid (POSIX) priority value (0-255, with zero being the lowest).
- **Access Routines**: `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()`
- **Dynamic Access Routines**: `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()`

Note that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see **4.14.2 POSIX and VxWorks Priority Numbering**, p.199.

A priority ceiling is defined by the following conditions:

- Any pthread attempting to acquire a mutex, whose priority is higher than the ceiling, cannot acquire the mutex.
- Any pthread whose priority is lower than the ceiling value has its priority elevated to the ceiling value for the duration that the mutex is held.
- The pthread’s priority is restored to its previous value when the mutex is released.

### 4.13.2 Condition Variables

A pthread condition variable corresponds to an object that permits pthreads to synchronize on an event or state represented by the value of a variable. This is a more complicated type of synchronization than the one allowed by mutexes only. Its main advantage is that is allows for passive waiting (as opposed to active waiting or polling) on a change in the value of the variable. Condition variables are used in conjunction with mutexes (one mutex per condition variable). The routines that can be used to act directly on a condition variable and on the condition variable attribute object are listed in **Table 4-12** and **Table 4-13** (respectively).
Clock Selection

The only attribute supported for the condition variable object is the clock ID. By default the `pthread_cond_timedwait()` routine uses the `CLOCK_REALTIME` clock. The clock type can be changed with the `pthread_condattr_setclock()` routine. The accepted clock IDs are `CLOCK_REALTIME` (the default) and `CLOCK_MONOTONIC`. For information about POSIX clocks, see 4.8 POSIX Clocks and Timers, p.177.
4.14 POSIX and VxWorks Scheduling

VxWorks can be configured with either the traditional (native) VxWorks scheduler or with a POSIX thread scheduler. Neither can be used to schedule processes (RTPs). The only entities that can be scheduled in VxWorks are tasks and pthreads.

The VxWorks implementation of a POSIX thread scheduler is an enhancement of the traditional VxWorks scheduler that provides additional scheduling facilities for pthreads running in processes.

With either scheduler, VxWorks tasks and pthreads share a single priority range and the same global scheduling scheme. With the POSIX thread scheduler, however, pthreads running in processes may have individual (concurrent) scheduling policies. Note that VxWorks must be configured with the POSIX thread scheduler in order to run pthreads in processes.

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

Table 4-16 provides an overview of how scheduling works for tasks and pthreads, for each of the schedulers, in both the kernel and processes (RTPs). The key differences are the following:

- The POSIX thread scheduler provides POSIX scheduling support for threads running in processes.
- In all other cases, the POSIX thread scheduler schedules pthreads and tasks in the same (non-POSIX) manner as the traditional VxWorks scheduler. (There is a minor difference between how it handles tasks and pthreads whose priorities have been lowered; see Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities, p.205.)
- The traditional VxWorks scheduler cannot be used to schedule pthreads in processes. In fact, pthreads cannot be started in processes unless VxWorks is configured with the POSIX thread scheduler.

The information provided in Table 4-16 is discussed in detail in subsequent sections.
Table 4-16 Task and Pthread Scheduling in the Kernel and in Processes

<table>
<thead>
<tr>
<th>Execution Environment</th>
<th>POSIX Thread Scheduler</th>
<th>Traditional VxWorks Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tasks</td>
<td>Pthreads</td>
</tr>
<tr>
<td>Kernel</td>
<td>Priority-based</td>
<td>Same as task scheduling.</td>
</tr>
<tr>
<td></td>
<td>preemptive, or round-robin scheduling.</td>
<td>No concurrent scheduling policies.</td>
</tr>
<tr>
<td>Processes</td>
<td>Priority-based</td>
<td>POSIX FIFO, round-robin, sporadic, or other (system default).</td>
</tr>
<tr>
<td></td>
<td>preemptive, or round robin scheduling.</td>
<td>Concurrent scheduling policies available.</td>
</tr>
</tbody>
</table>

a. The traditional VxWorks scheduler cannot ensure behavioral compliance with the POSIX 1 standard.
4.14.1 Differences in POSIX and VxWorks Scheduling

In general, the POSIX scheduling model and scheduling in a VxWorks system differ in the following ways—regardless of whether the system is configured with the Wind River POSIX thread scheduler or the traditional VxWorks scheduler:

- POSIX supports a two-level scheduling model that includes the concept of contention scope, by which the scheduling of pthreads can apply system wide or on a process basis. In VxWorks, on the other hand, processes (RTPs) cannot themselves be scheduled, and tasks and pthreads are scheduled on a system-wide (kernel and processes) basis.

- POSIX applies scheduling policies on a process-by-process and thread-by-thread basis. VxWorks applies scheduling policies on a system-wide basis, for all tasks and pthreads, whether in the kernel or in processes. This means that all tasks and pthreads use either a preemptive priority scheme or a round-robin scheme. The only exception to this rule is that pthreads executing in processes can be subject to concurrent (individual) scheduling policies, including sporadic scheduling (note that the POSIX thread scheduler must be used in this case).

- POSIX supports the concept of scheduling allocation domain; that is, the association between processes or threads and processors. Since VxWorks does not support multi-processor hardware, there is only one domain on VxWorks and all the tasks and pthreads are associated to it.

- The POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see 4.14.2 POSIX and VxWorks Priority Numbering, p. 199.

- VxWorks does not support the POSIX thread-concurrency feature, as all threads are scheduled. The POSIX thread-concurrency APIs are provided for application portability, but they have no effect.

4.14.2 POSIX and VxWorks Priority Numbering

The POSIX priority numbering scheme is the inverse of the VxWorks priority numbering scheme. In POSIX, the higher the number, the higher the priority. In VxWorks, the lower the number, the higher the priority, where 0 is the highest priority.

The priority numbers used with the POSIX scheduling library, schedPxClib, do not, therefore, match those used and reported by all other components of VxWorks. You can change the default POSIX numbering scheme by setting the global variable posixPriorityNumbering to FALSE. If you do so, schedPxClib uses the
VxWorks numbering scheme (a smaller number means a higher priority) and its priority numbers match those used by the other components of VxWorks.

In the following sections, discussions of pthreads and tasks at the same priority level refer to functionally equivalent priority levels, and not to priority numbers.

4.14.3 Default Scheduling Policy

All VxWorks tasks and pthreads are scheduled according to the system-wide default scheduling policy. The only exception to this rule is for pthreads running in user mode (in processes). In this case, concurrent scheduling policies that differ from the system default can be applied to pthreads.

Note that pthreads can be run in processes only if VxWorks is configured with the POSIX thread scheduler; they cannot be run in processes if VxWorks is configured with the traditional scheduler.

The system-wide default scheduling policy for VxWorks, regardless of which scheduler is used, is priority-based preemptive scheduling—which corresponds to the POSIX\texttt{SCHED\_FIFO} scheduling policy.

At run-time the active system-wide default scheduling policy can be changed to round-robin scheduling with the \texttt{kernelTimeSlice()} routine. It can be changed back by calling \texttt{kernelTimeSlice()} with a parameter of zero. VxWorks round-robin scheduling corresponds to the POSIX\texttt{SCHED\_RR} policy.

The \texttt{kernelTimeSlice()} routine cannot be called in user mode (that is, from a process). A call with a non-zero parameter immediately affects all kernel and user tasks, all kernel pthreads, and all user pthreads using the \texttt{SCHED\_OTHER} policy. Any user pthreads running with the \texttt{SCHED\_RR} policy are unaffected by the call; but those started after it use the newly defined timeslice.
4.14.4 VxWorks Traditional Scheduler

The VxWorks traditional scheduler can be used with both tasks and pthreads in the kernel. It cannot be used with pthreads in processes. If VxWorks is configured with the traditional scheduler, a `pthread_create()` call in a process fails and the errno is set to ENOSYS.

The traditional VxWorks scheduler schedules pthreads as if they were tasks. All tasks and pthreads executing in a system are therefore subject to the current default scheduling policy (either the priority-based preemptive policy or the round-robin scheduling policy; see 4.14.3 Default Scheduling Policy, p.200), and concurrent policies cannot be applied to individual pthreads. For general information about the traditional scheduler and how it works with tasks, see 3.2.2 Task Scheduling, p.83.

The scheduling options provided by the traditional VxWorks scheduler are similar to the POSIX ones. The following pthreads scheduling policies correspond to the traditional VxWorks scheduling policies:

- **SCHED_FIFO** is similar to VxWorks priority-based preemptive scheduling. There are differences as to where tasks or pthreads are placed in the ready queue if their priority is lowered; see Caveats About Scheduling Behavior with the POSIX Thread Scheduler, p.204.

- **SCHED_RR** corresponds to VxWorks round-robin scheduling.

- **SCHED_OTHER** corresponds to the current system-wide default scheduling policy. The SCHED_OTHER policy is the default policy for pthreads in VxWorks.

There is no VxWorks traditional scheduler policy that corresponds to SCHED_SPORADIC.

**Configuring VxWorks with the Traditional Scheduler**

VxWorks is configured with the traditional scheduler by default. This scheduler is provided by the INCLUDE_VX_TRADITIONAL_SCHEDULER component.

**Caveats About Scheduling Behavior with the VxWorks Traditional Scheduler**

Concurrent scheduling policies are not supported for pthreads in the kernel, and care must therefore be taken with pthread scheduling-inheritance and scheduling policy attributes.

If the scheduling-inheritance attribute is set to PTHREAD_EXPLICIT_SCHED and the scheduling policy to SCHED_FIFO or SCHED_RR, and this policy does not
match the current system-wide default scheduling policy, the creation of pthreads fails.

Wind River therefore recommends that you always use **PTHREAD_INHERIT_SCHED** (which is the default) as a scheduling-inheritance attribute. In this case the current VxWorks scheduling policy applies, and the parent pthread’s priority is used. Or, if the pthread must be started with a different priority than its parent, the scheduling-inheritance attribute can be set to **PTHREAD_EXPLICIT_SCHED** and the scheduling policy attribute set to be **SCHED_OTHER** (which corresponds to the current system-wide default scheduling policy).

In order to take advantage of the POSIX scheduling model, VxWorks must be configured with the POSIX thread scheduler, and the pthreads in question must be run in processes (RTPs). See 4.14.5 **POSIX Threads Scheduler**, p.202.

### 4.14.5 POSIX Threads Scheduler

The POSIX thread scheduler can be used to schedule both pthreads and tasks in a VxWorks system. Note that the purpose of the POSIX thread scheduler is to provide POSIX scheduling support for pthreads running in processes. There is no reason to use it in a system that does not require this support (kernel-only systems, or systems with processes but without pthreads).

The POSIX thread scheduler is required for running pthreads in processes, where it provides compliance with POSIX 1003.1 for pthread scheduling (including concurrent scheduling policies). If VxWorks is not configured with the POSIX thread scheduler, pthreads cannot be created in processes.

**NOTE:** The POSIX priority numbering scheme is the inverse of the VxWorks scheme, so references to *a given priority level* or *same level* in comparisons of these schemes refer to functionally equivalent priority levels, and not to priority numbers. For more information about the numbering schemes see 4.14.2 **POSIX and VxWorks Priority Numbering**, p.199.

#### Scheduling in the Kernel

The POSIX thread scheduler schedules *kernel tasks* and *kernel pthreads* in the same manner as the traditional VxWorks task scheduler. See 3.2.2 **Task Scheduling**, p.83 for information about the traditional scheduler and how it works with VxWorks tasks, and 4.14.4 **VxWorks Traditional Scheduler**, p.201 for information about how
POSIX scheduling policies correspond to the traditional VxWorks scheduling policies.

**Scheduling in Processes**

When VxWorks is configured with the POSIX thread scheduler, *tasks* executing in processes are scheduled according to system-wide default scheduling policy. On the other hand, *pthreads* executing in processes are scheduled according to POSIX 1003.1. Scheduling policies can be assigned to each pthread and changed dynamically. The scheduling policies are as follows:

- **SCHED_FIFO** is a preemptive priority scheduling policy. For a given priority level, pthreads scheduled with this policy are handled as peers of the VxWorks tasks at the same level. There is a slight difference in how pthreads and tasks are handled if their priorities are lowered (for more information; see *Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities*, p.205).

- **SCHED_RR** is a per-priority round-robin scheduling policy. For a given priority level, all pthreads scheduled with this policy are given the same time of execution (time-slice) before giving up the CPU.

- **SCHED_SPORADIC** is a policy used for aperiodic activities, which ensures that the pthreads associated with the policy are served periodically at a high priority for a bounded amount of time, and a low background priority at all other times.

- **SCHED_OTHER** corresponds to the scheduling policy currently in use for VxWorks tasks, which is either preemptive priority or round-robin. Pthreads scheduled with this policy are submitted to the system’s global scheduling policy, exactly like VxWorks tasks or kernel pthreads.

Note the following with regard to the VxWorks implementation of the **SCHED_SPORADIC** policy:

- The system periodic clock is used for time accounting.

- Dynamically changing the scheduling policy to **SCHED_SPORADIC** is not supported; however, dynamically changing the policy from **SCHED_SPORADIC** to another policy is supported.

- VxWorks does not impose an upper limit on the maximum number of replenishment events with the **SS_REPL_MAX** macro. A default of 40 events is set with the **sched_ss_max_repl** field of the thread attribute structure, which can be changed.
Configuring VxWorks with the POSIX Thread Scheduler

To configure VxWorks with the POSIX thread scheduler, add the INCLUDE_POSIX_PTHREAD_SCHEDULER component to the kernel.

Note that only the SCHED_FIFO, SCHED_RR, and SCHED_OTHER scheduling policies are provided with the INCLUDE_POSIX_PTHREAD_SCHEDULER component. For the SCHED_SPORADIC scheduling policy, the INCLUDE_PX_SCHED_SPORADIC_POLICY component must be included as well. The bundle BUNDLE_RTP_POSIX_PSE52 includes the INCLUDE_PX_SCHED_SPORADIC_POLICY component as well as the INCLUDE_POSIX_PTHREAD_SCHEDULER component.

The configuration parameter POSIX_PTHREAD_RR_TIMESLICE may be used to configure the default time slicing interval for pthreads started with the SCHED_RR policy. To modify the time slice at run time, call kernelTimeSlice() with a different time slice value. The new time slice value only affects pthreads created after the kernelTimeSlice() call.

NOTE: The INCLUDE_POSIX_PTHREAD_SCHEDULER component is a standalone component. It is not dependent on any other POSIX components nor is it automatically included with any other components.

The POSIX thread scheduler must be added explicitly with either the INCLUDE_POSIX_PTHREAD_SCHEDULER component or the BUNDLE_RTP_POSIX_PSE52 bundle.

The POSIX thread scheduler component is independent because it is intended to be used only with pthreads in processes; kernel-only systems that use pthreads, have no need to change from the default VxWorks traditional scheduler.

Caveats About Scheduling Behavior with the POSIX Thread Scheduler

Using the POSIX thread scheduler involves a few complexities that should be taken into account when designing your system. Care should be taken with regard to the following:

- Using both round-robin and priority-based preemptive scheduling policies.
- Running pthreads with the individual SCHED_OTHER policy.
- Differences in re-queuing pthreads and tasks with lowered priorities.
- Backwards compatibility issues for POSIX applications designed for the VxWorks traditional scheduler.
Using both Round-Robin and Priority-Based Preemptive Policies

Using a combination of round-robin and priority-based preemptive policies for tasks and pthreads of the same priority level can lead to task or pthread CPU starvation for the entities running with the round-robin policy.

When VxWorks is running with round-robin scheduling as the system default, tasks may not run with their expected time slice if there are pthreads running at the same priority level with the concurrent (individual) SCHED_FIFO policy. This is because one of the pthreads may monopolize the CPU and starve the tasks. Even if the usurper pthread is preempted, it stays at the head of its priority lists (as POSIX mandates), and continues to monopolize the CPU when that priority level can run again. Pthreads scheduled with the SCHED_RR or SCHED_OTHER policy are at the same disadvantage as the tasks scheduled with the round-robin policy.

Similarly, when VxWorks is running with preemptive scheduling as the system default, tasks may starve pthreads with the same priority level if the latter have the concurrent (individual) SCHED_RR policy.

Running pthreads with the Concurrent SCHED_OTHER Policy

Pthreads created with the concurrent (individual) SCHED_OTHER policy behave the same as the system-wide default scheduling policy, which means that:

- If the system default is currently priority-based preemptive scheduling, the SCHED_OTHER pthreads run with the preemptive policy.
- If the system default is currently round-robin scheduling, the SCHED_OTHER pthreads run with the round-robin policy.

While changing the default system policy from priority-based preemptive scheduling to round-robin scheduling (or the opposite) changes the effective scheduling policy for pthreads created with SCHED OTHER, it has no effect on pthreads created with SCHED_RR or SCHED_FIFO.

Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities

The POSIX thread scheduler re-queues pthreads that have had their priority lowered differently than it re-queues tasks that have had their priority lowered. The difference is as follows:

- When the priority of a pthread is lowered (with the pthread_setschedpri() routine), the POSIX thread scheduler places it at the head of the priority list.
- When the priority of a task is lowered (with the taskPrioritySet() routine), the POSIX thread scheduler places it at the tail of the priority list—which is the same as what the traditional VxWorks scheduler would do.
What this means is that lowering the priority of a task and a pthread may have a different effect on when they will run (if there are other tasks or pthreads in their priority list). For example, if a task and a pthread each have their priority lowered to effectively the same level, the pthread will be at the head of the priority list and the task at the end. The pthread will run before any other pthreads or tasks at this level, and the task after any other pthreads or tasks.

**Backwards Compatibility Issues for Applications**

Using the POSIX thread scheduler changes the behavior of POSIX applications that were written to run with the traditional VxWorks scheduler. For existing POSIX applications that require backward-compatibility, the scheduling policy can be changed to `SCHED_OTHER` for all pthreads. This causes their policy to default to the active VxWorks task scheduling policy (as was the case before the introduction of the POSIX thread scheduler).

### 4.14.6 POSIX Scheduling Routines

The POSIX 1003.1b scheduling routines provided by the `schedPxLib` library for VxWorks are described in Table 4-17.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sched_get_priority_max()</code></td>
<td>Gets the maximum pthread priority.</td>
</tr>
<tr>
<td><code>sched_get_priority_min()</code></td>
<td>Gets the minimum pthread priority.</td>
</tr>
<tr>
<td><code>sched_rr_get_interval()</code></td>
<td>If round-robin scheduling is in effect, gets the time slice length.</td>
</tr>
<tr>
<td><code>sched_yield()</code></td>
<td>Relinquishes the CPU.</td>
</tr>
</tbody>
</table>

For more information about these routines, see the `schedPxLib` API reference.

Note that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see 4.14.2 POSIX and VxWorks Priority Numbering, p.199.

To include the `schedPxLib` library in the system, configure VxWorks with the `INCLUDE_POSIX_SCHED` component.
Process-based (RTP) applications are automatically linked with the schedPxLib library when they are compiled.

4.14.7 Getting Scheduling Parameters: Priority Limits and Time Slice

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

User tasks and pthreads can use sched_rr_get_interval() to determine the length of the current time-slice interval. This routine takes as an argument a pointer to a timespec structure (defined in time.h), and writes the number of seconds and nanoseconds per time slice to the appropriate elements of that structure.

Note that a non-null result does not imply that the POSIX thread calling this routine is being scheduled with the SCHED_RR policy. To make this determination, a pthread must use the pthread_getschedparam() routine.

Example 4-5  Getting the POSIX Round-Robin Time Slice

```c
/*
 * The following example gets the length of the time slice,
 * and then displays the time slice.
 */

/* includes */
#include <time.h>
#include <sched.h>

STATUS rrgetintervalTest (void)
{
    struct timespec slice;
    if (sched_rr_get_interval (0, &slice) == ERROR)
    {
        printf ("get-interval test failed\n");
        return (ERROR);
    }
    printf ("time slice is %ld seconds and %ld nanoseconds\n",
            slice.tv_sec, slice.tv_nsec);
    return (OK);
}
```
4.15 POSIX Semaphores

POSIX defines both named and unnamed semaphores, which have the same properties, but which use slightly different interfaces. The POSIX semaphore library provides routines for creating, opening, and destroying both named and unnamed semaphores.

When opening a named semaphore, you assign a symbolic name, which the other named-semaphore routines accept as an argument. The POSIX semaphore routines provided by `semPxLib` are shown in Table 4-18.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_init()</code></td>
<td>Initializes an unnamed semaphore.</td>
</tr>
<tr>
<td><code>sem_destroy()</code></td>
<td>Destroys an unnamed semaphore.</td>
</tr>
<tr>
<td><code>sem_open()</code></td>
<td>Initializes/opens a named semaphore.</td>
</tr>
<tr>
<td><code>sem_close()</code></td>
<td>Closes a named semaphore.</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Removes a named semaphore.</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>Lock a semaphore.</td>
</tr>
<tr>
<td><code>sem_trywait()</code></td>
<td>Lock a semaphore only if it is not already locked.</td>
</tr>
<tr>
<td><code>sem_post()</code></td>
<td>Unlock a semaphore.</td>
</tr>
<tr>
<td><code>sem_getvalue()</code></td>
<td>Get the value of a semaphore.</td>
</tr>
<tr>
<td><code>sem_timedwait()</code></td>
<td>Lock a semaphore with a timeout.</td>
</tr>
</tbody>
</table>

Note that the behavior of the user space `sem_open()` routine complies with the POSIX specification, and thus differs from the kernel version of the routine. The kernel version of `sem_open()` returns a reference copy for the same named semaphore when called multiple times, provided that `sem_unlink()` is not called.

2. Some operating systems, such as UNIX, require symbolic names for objects that are to be shared among processes. This is because processes do not normally share memory in such operating systems. In VxWorks, named semaphores can be used to share semaphores between real-time processes. In the VxWorks kernel there is no need for named semaphores, because all kernel objects have unique identifiers. However, using named semaphores of the POSIX variety provides a convenient way of determining the object’s ID.
The user space version of `sem_open()` returns the same ID for the same named semaphore when called multiple times.

To include the POSIX semPxLib library semaphore routines in the system, configure VxWorks with the INCLUDE_POSIX_SEM component.

VxWorks also provides `semPxLibInit()`, a non-POSIX (kernel-only) routine that initializes the kernel’s POSIX semaphore library. It is called by default at boot time when POSIX semaphores have been included in the VxWorks configuration.

Process-based (RTP) applications are automatically linked with the `semPxLib` library when they are compiled. The library is automatically initialized when the process starts.

### 4.15.1 Comparison of POSIX and VxWorks Semaphores

POSIX semaphores are *counting* semaphores; that is, they keep track of the number of times they are given. The VxWorks semaphore mechanism is similar to that specified by POSIX, except that VxWorks semaphores offer these additional features:

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

When these features are important, VxWorks semaphores are preferable to POSIX semaphores. (For information about these features, see 3. Multitasking.)

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

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

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

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

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

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

When a semaphore is used for mutual exclusion, it is typically initialized to a value greater than zero, meaning that the resource is available. Therefore, the first task to lock the semaphore does so without blocking, setting the semaphore to 0 (locked). Subsequent tasks will block until the semaphore is released. As with the previous scenario, when the semaphore is released the task with the highest priority is unblocked.

When used in a user application, unnamed semaphores can be accessed only by the tasks belonging to the process executing the application.

Only public named semaphores (that is, named semaphores whose name begins with a forward slash) can be shared between processes or between processes and the kernel. For information about public objects, see 3.3.1 Public and Private Objects, p.105.
Example 4-6  POSIX Unnamed Semaphores

/*
 * This example uses unnamed semaphores to synchronize an action between the
 * calling task and a task that it spawns (tSyncTask). To run from the
 * shell,
 * spawn as a task:
 * -> sp unnameSem
 */
/* includes */
#include <vxWorks.h>
#include <semaphore.h>
/* forward declarations */
void syncTask (sem_t * pSem);
/*************************************************************************
* unnameSem - test case for unamed semaphores
* This routine tests unnamed semaphores.
* RETURNS: N/A
* ERRNOS: N/A
*/
void unnameSem (void)
{
    sem_t * pSem;
    /* reserve memory for semaphore */
    pSem = (sem_t *) malloc (sizeof (sem_t));
    if (pSem == NULL)
        {
            printf ("pSem allocation failed\n");
            return;
        }
    /* initialize semaphore to unavailable */
    if (sem_init (pSem, 0, 0) == -1)
        {
            printf ("unnameSem: sem_init failed\n");
            free ((char *) pSem);
            return;
        }
    /* create sync task */
    printf ("unnameSem: spawning task...\n");
if (taskSpawn ("tSyncTask", 90, 0, 2000, syncTask, pSem) == ERROR)
{
    printf ("Failed to spawn tSyncTask\n");
    sem_destroy (pSem);
    free ((char *) pSem);
    return;
}

/* do something useful to synchronize with syncTask */
/* unlock sem */
printf ("unnameSem: posting semaphore - synchronizing action\n");
if (sem_post (pSem) == -1)
{
    printf ("unnameSem: posting semaphore failed\n");
    sem_destroy (pSem);
    free ((char *) pSem);
    return;
}

/* all done - destroy semaphore */
if (sem_destroy (pSem) == -1)
{
    printf ("unnameSem: sem_destroy failed\n");
    return;
}
free ((char *) pSem);
}

void syncTask
{
    sem_t * pSem
}

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

/* do something useful here */
}
4.15.3 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 4-19**.

<table>
<thead>
<tr>
<th>Flag Settings</th>
<th>If Semaphore Exists</th>
<th>If Semaphore Does Not Exist</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Semaphore is opened.</td>
<td>Routine fails.</td>
</tr>
<tr>
<td><strong>O_CREAT</strong></td>
<td>Semaphore is opened.</td>
<td>Semaphore is created.</td>
</tr>
<tr>
<td><strong>O_CREAT</strong> and <strong>O_EXCL</strong></td>
<td>Routine fails.</td>
<td>Semaphore is created.</td>
</tr>
<tr>
<td><strong>O_EXCL</strong></td>
<td>Routine fails.</td>
<td>Routine fails.</td>
</tr>
</tbody>
</table>

Once initialized, a semaphore remains usable until explicitly destroyed. Tasks can explicitly mark a semaphore for destruction at any time, but the system only destroys the semaphore when no task has the semaphore open.

If VxWorks is configured with `INCLUDE_POSIX_SEM_SHOW`, you can use `show()` from the shell (with the C interpreter) to display information about a POSIX semaphore.  

3. The `show()` routine is not a POSIX routine, nor is it meant to be used programmatically. It is designed for interactive use with the shell (with the shell's C interpreter).
This example shows information about the POSIX semaphore `mySem` with two tasks blocked and waiting for it:

```
-> show semId
  value = 0 = 0x0
  Semaphore name : mySem
  sem_open() count : 3
  Semaphore value : 0
  No. of blocked tasks : 2
```

Note that `show()` takes the semaphore ID as the argument.

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

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

**NOTE:** POSIX named semaphores may be shared between processes only if their names start with a `/` (forward slash) character. They are otherwise private to the process in which they were created, and cannot be accessed from another process. See 3.3.1 Public and Private Objects, p.105.
Example 4-7  POSIX Named Semaphores

/*
* In this example, nameSem() creates a task for synchronization. The
* new task, tSyncSemTask, blocks on the semaphore created in nameSem().
* Once the synchronization takes place, both tasks close the semaphore,
* and nameSem unlinks it. To run this task from the shell, spawn
* nameSem as a task:
* -> sp nameSem, "myTest"
*/
/* includes */
#include <vxWorks.h>
#include <taskLib.h>
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>
/* forward declaration */
void syncSemTask (char * name);

/*********************************************/
* nameSem - test program for POSIX semaphores
* This routine opens a named semaphore and spawns a task, tSyncSemTask, which
* waits on the named semaphore.
* RETURNS: N/A
* ERRNO: N/A
*/
void nameSem
(char * name)
{
    sem_t * semId;
    /* create a named semaphore, initialize to 0*/
    printf ("nameSem: creating semaphore\n");
    if ((semId = sem_open (name, O_CREAT, 0, 0)) == (sem_t *) -1)
    {
        printf ("nameSem: sem_open failed\n");
        return;
    }

    printf ("nameSem: spawning sync task\n");
    if (taskSpawn ("tSyncSemTask", 90, 0, 4000, (FUNCPTR) syncSemTask,
        (int) name, 0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
        printf ("nameSem: unable to spawn tSyncSemTask\n");
        sem_close(semId);
    }
}
return;
}
/* do something useful to synchronize with syncSemTask */

/* give semaphore */
printf("nameSem: posting semaphore - synchronizing action\n");
if (sem_post (semId) == -1)
{
    printf("nameSem: sem_post failed\n");
    sem_close(semId);
    return;
}
/* all done */
if (sem_close (semId) == -1)
{
    printf("nameSem: sem_close failed\n");
    return;
}
if (sem_unlink (name) == -1)
{
    printf("nameSem: sem_unlink failed\n");
    return;
}
printf("nameSem: closed and unlinked semaphore\n");
}/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%***/

/* syncSemTask - waits on a named POSIX semaphore */
/* This routine waits on the named semaphore created by nameSem(). */
/* RETURNS: N/A */
/* ERRNO: N/A */
void syncSemTask
{
    char * name
    {
        sem_t * semId;

        /* open semaphore */
        printf("syncSemTask: opening semaphore\n");
        if ((semId = sem_open (name, 0)) == (sem_t *) -1)
        {
            printf("syncSemTask: sem_open failed\n");
            return;
        }
/* block waiting for synchronization from nameSem */
printf("syncSemTask: attempting to take semaphore...\n");
if (sem_wait (semId) == -1) {
    printf("syncSemTask: taking sem failed\n");
    return;
}
printf("syncSemTask: has semaphore, doing sync'ed action ...\n");
/* do something useful here */
if (sem_close (semId) == -1) {
    printf("syncSemTask: sem_close failed\n");
    return;
}
}
4.16 POSIX Message Queues

The POSIX message queue routines, provided by mqPxBib, are shown in Table 4-20.

Table 4-20 POSIX Message Queue Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mq_open()</td>
<td>Opens a message queue.</td>
</tr>
<tr>
<td>mq_close()</td>
<td>Closes a message queue.</td>
</tr>
<tr>
<td>mq_unlink()</td>
<td>Removes a message queue.</td>
</tr>
<tr>
<td>mq_send()</td>
<td>Sends a message to a queue.</td>
</tr>
<tr>
<td>mq_receive()</td>
<td>Gets a message from a queue.</td>
</tr>
<tr>
<td>mq_notify()</td>
<td>Signals a task that a message is waiting on a queue.</td>
</tr>
<tr>
<td>mq_setattr()</td>
<td>Sets a queue attribute.</td>
</tr>
<tr>
<td>mq_getattr()</td>
<td>Gets a queue attribute.</td>
</tr>
<tr>
<td>mq_timedsend()</td>
<td>Sends a message to a queue, with a timeout.</td>
</tr>
<tr>
<td>mq_timedreceive()</td>
<td>Gets a message from a queue, with a timeout.</td>
</tr>
</tbody>
</table>

Note that there are behavioral differences between the kernel and user space versions of mq_open(). The kernel version allows for creation of a message queue for any permission specified by the oflags parameter. The user-space version complies with the POSIX PSE52 profile, so that after the first call, any subsequent calls in the same process are only allowed if an equivalent or lower permission is specified.
Table 4-21 describes the permissions that are allowed.

<table>
<thead>
<tr>
<th>Permission When Created</th>
<th>Access Permitted in Same Process</th>
<th>Access Forbidden</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_RDONLY</td>
<td>O_RDONLY</td>
<td>O_RDWR, O_WRONLY</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>O_WRONLY</td>
<td>O_RDWR, O_RDONLY</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>O_RDWR, O_WRONLY, O_RDONLY</td>
<td></td>
</tr>
</tbody>
</table>

Process-based (RTP) applications are automatically linked with the `mqPxLib` library when they are compiled. Initialization of the library is automatic as well, when the process is started.

For information about the VxWorks message queue library, see the `msgQLib` API reference.
4.16.1 Comparison of POSIX and VxWorks Message Queues

POSIX message queues are similar to VxWorks message queues, except that POSIX message queues provide messages with a range of priorities. The differences are summarized in Table 4-22.

<table>
<thead>
<tr>
<th>Feature</th>
<th>VxWorks Message Queues</th>
<th>POSIX Message Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Message Queue Levels</td>
<td>1 (specified by MSG_PRI_NORMAL</td>
<td>32 (specified by MAX_PRIO_MAX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(specified by MAX_PRIO_MAX)</td>
</tr>
<tr>
<td>Blocked Message Queues</td>
<td>FIFO or priority-based</td>
<td>Priority-based</td>
</tr>
<tr>
<td>Received with Timeout</td>
<td>msgQReceive( ) option</td>
<td>mq_timedreceive( ) (user-space only)</td>
</tr>
<tr>
<td>Task Notification</td>
<td>With VxWorks message queue events</td>
<td>mq_notify( )</td>
</tr>
<tr>
<td>Close/Unlink Semantics</td>
<td>With msgQOpen library</td>
<td>Yes</td>
</tr>
<tr>
<td>Send with Timeout</td>
<td>msgQsend( ) option</td>
<td>mq_timesend( ) (user-space only)</td>
</tr>
</tbody>
</table>

4.16.2 POSIX Message Queue Attributes

A POSIX message queue has the following attributes:

- an optional O_NONBLOCK flag, which prevents a mq_receive( ) call from being a blocking call if the message queue is empty
- the maximum number of messages in the message queue
- the maximum message size
- the number of messages currently on the queue

Tasks can set or clear the O_NONBLOCK flag using mq_setattr( ), and get the values of all the attributes using mq_getattr(). (As allowed by POSIX, this implementation of message queues makes use of a number of internal flags that are not public.)
Example 4-8  Setting and Getting Message Queue Attributes

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

/* includes */
#include <vxWorks.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>

/* defines */
#define MSG_SIZE 16

int attrEx
{
    char * name

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

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

    /* change attributes on queue - turn on non-blocking */
    attr.mq_flags = O_NONBLOCK;
    if (mq_setattr (mqPXId, &attr, &oldAttr) == -1)
        return (ERROR);
    else
    { /* paranoia check - oldAttr should not include non-blocking. */
        if (oldAttr.mq_flags & O_NONBLOCK)
            return (ERROR);
        else
            printf("mq_setattr turning on non-blocking succeeded\n");
    }
}
/* try receiving - there are no messages but this shouldn’t block */
if (mq_receive (mqPXId, buffer, MSG_SIZE, &prio) == -1)
{
    if (errno != EAGAIN)
        return (ERROR);
    else
        printf("mq_receive with non-blocking didn’t block on empty queue\n");
}
else
    return (ERROR);

/* use mq_getattr to verify success */
if (mq_getattr (mqPXId, &oldAttr) == -1)
    return (ERROR);
else
{
    /* test that we got the values we think we should */
    if (!(oldAttr.mq_flags & O_NONBLOCK) || (oldAttr.mq_curmsgs != 0))
        return (ERROR);
    else
        printf("queue attributes are:\n\tblocking is %s\n\tmessage size is: %d\n\tmax messages in queue: %d\n\tno. of current msgs in queue: %d",
            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);

4.16.3 Communicating Through a Message Queue

Before a set of tasks can communicate through a POSIX message queue, one of the
tasks must create the message queue by calling \texttt{mq\_open()} with the \texttt{O\_CREATE} 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
\texttt{O\_CREATE} flag; subsequent tasks can open the queue for receiving only
(\texttt{O\_RDONLY}), sending only (\texttt{O\_WONLY}), or both sending and receiving
(\texttt{O\_RDWR}).
To put messages on a queue, use `mq_send()`. If a task attempts to put a message on the queue when the queue is full, the task blocks until some other task reads a message from the queue, making space available. To avoid blocking on `mq_send()`, set `O_NONBLOCK` when you open the message queue. In that case, when the queue is full, `mq_send()` returns -1 and sets `errno` to `EAGAIN` instead of pending, allowing you to try again or take other action as appropriate.

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

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

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

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

**NOTE:** In VxWorks, a POSIX message queue whose name does not start with a forward-slash (/) character is considered private to the process that has opened it and can not be accessed from another process. A message queue whose name starts with a forward-slash (/) character is a public object, and other processes can access it (as according to the POSIX standard). See 3.3.1 Public and Private Objects, p.105.
Example 4-9  POSIX Message Queues

/*
 * In this example, the mqExInit() routine spawns two tasks that
 * communicate using the message queue.
 * To run this test case on the target shell:
 *  -> sp mqExInit
 */

/* mqEx.h - message example header */
/* defines */
#define MQ_NAME "exampleMessageQueue"
/* forward declarations */
void receiveTask (void);
void sendTask (void);

/* testMQ.c - example using POSIX message queues */
/* includes */
#include <vxWorks.h>
#include <taskLib.h>
#include <stdio.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>
#include <mqEx.h>
/* defines */
#define HI_PRIO 31
#define MSG_SIZE 16
#define MSG "greetings"

/* mqExInit - main for message queue send and receive test case */
/* This routine spawns to tasks to perform the message queue send and receive */
/* test case. */
/* RETURNS: OK, or ERROR */
/* ERRNOS: N/A */
int mqRxInit (void)
{
    /* create two tasks */

    if (taskSpawn ("tRcvTask", 151, 0, 4000, 
        (FUNCPTR) receiveTask, 
        0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR) 
    {
        printf ("taskSpawn of tRcvTask failed\n");
        return (ERROR);
    }

    if (taskSpawn ("tSndTask", 152, 0, 4000, 
        (FUNCPTR) sendTask, 
        0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR) 
    {
        printf ("taskSpawn of tSndTask failed\n");
        return (ERROR);
    }

    return (OK);
}

/***************************************************************************/
/*
* receiveTask - receive messages from the message queue
* This routine creates a message queue and calls mq_receive() to wait for
* a message arriving in the message queue.
* RETURNS: OK, or ERROR
* ERRNOS: N/A
*/
/***************************************************************************/
void receiveTask (void)
{
    mqd_t mqPXId; /* msg queue descriptor */
    char msg[MSG_SIZE]; /* msg buffer */
    int prio; /* priority of message */

    /* open message queue using default attributes */

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

    /* try reading from queue */

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

/**************************************************************************
* sendTask - send a message to a message queue
* This routine opens an already created message queue and
* calls mq_send() to send a message to the opened message queue.
* RETURNS: OK, or ERROR
* ERRNOS: N/A
*/
void sendTask (void)
{
    mqd_t mqPXId; /* msg queue descriptor */

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

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

4.16.4 Notification of Message Arrival

A pthread (or task) can use the mq_notify() routine to request notification of the
arrival of a message at an empty queue. The pthread can thereby avoid blocking or
polling to wait for a message.

Each queue can register only one pthread for notification at a time. Once a queue
has a pthread to notify, no further attempts to register with mq_notify() can
succeed until the notification request is satisfied or cancelled.
Once a queue sends notification to a pthread, the notification request is satisfied, and the queue has no further special relationship with that particular pthread; that is, the queue sends a notification signal only once for each \texttt{mq_notify()} request. To arrange for one specific pthread to continue receiving notification signals, the best approach is to call \texttt{mq_notify()} from the same signal handler that receives the notification signals.

To cancel a notification request, specify \texttt{NULL} instead of a notification signal. Only the currently registered pthread can cancel its notification request.

The \texttt{mq_notify()} mechanism does not send notification:

- When additional messages arrive at a message queue that is not empty. That is, notification is only sent when a message arrives at an empty message queue.
- If another pthread was blocked on the queue with \texttt{mq_receive()}. 
- After a response has been made to the call to \texttt{mq_notify()}. That is, only one notification is sent per \texttt{mq_notify()} call.

### Example 4-10  Message Queue Notification

```c
/*
 * In this example, a task uses mq_notify() to discover when a message
 * has arrived on a previously empty queue. To run this from the shell:
 * -> ld < mq_notify_test.o
 * -> sp exMqNotify, "greetings"
 * -> mq_send
 * /

/* includes */
#include <vxWorks.h>
#include <signal.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

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

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

To cancel a notification request, specify \texttt{NULL} instead of a notification signal. Only the currently registered pthread can cancel its notification request.
int exMqNotify
{
    struct mq_attr attr;     /* queue attribute structure */
    struct sigevent sigNotify;     /* to attach notification */
    struct sigaction mySigAction; /* to attach signal handler */
    mqd_t exMqId;         /* id of message queue */
    int cnt = 0;
    /* Minor sanity check; avoid exceeding msg buffer */
    if (MSG_SIZE <= strlen (pMessage))
    {
        printf ("exMqNotify: message too long\n");
        return (-1);
    }
    /* Install signal handler for the notify signal and fill in
    * a sigaction structure and pass it to sigaction(). Because the handler
    * needs the siginfo structure as an argument, the SA_SIGINFO flag is
    * set in sa_flags.
    */
    mySigAction.sa_sigaction = exNotificationHandle;
    mySigAction.sa_flags = SA_SIGINFO;
    sigemptyset (&mySigAction.sa_mask);

    if (sigaction (SIGUSR1, &mySigAction, NULL) == -1)
    {
        printf ("sigaction failed\n");
        return (-1);
    }
    /* Create a message queue - fill in a mq_attr structure with the
    * size and no. of messages required, and pass it to mq_open().
    */
    attr.mq_flags = 0;
    attr.mq_maxmsg = 2;
    attr.mq_msgsize = MSG_SIZE;
if ((exMqId = mq_open (QNAM, O_CREAT | O_RDWR | O_NONBLOCK, 0, &attr))
    == (mqd_t) - 1 )
{
    printf ("mq_open failed\n");
    return (-1);
}

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

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

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

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

    exMqRead (exMqId);

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

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

    /* Cleanup */

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


}  

/* More cleanup */

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

return (0);
}

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

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

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

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

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

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

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

        /*
        * Read in the messages - uses a loop to read in the messages
        * because a notification is sent ONLY when a message is sent on
        * an EMPTY message queue. There could be multiple msgs if, for
        * example, a higher-priority task was sending them. Because the
        * message queue was opened with the O_NONBLOCK flag, eventually
        * this loop exits with errno set to EAGAIN (meaning we did an
        * mq_receive() on an empty message queue).
        */
        while (mq_receive (exMqId, msg, MSG_SIZE, &prio) != -1)
        {
            printf ("exMqRead: mqId (0x%x) received message: %s\n", exMqId, msg);
        }
        if (errno != EAGAIN)
        {
            printf ("mq_receive: errno = %d\n", errno);
        }
}
4.17 POSIX Signals

VxWorks provides POSIX signal routines in user space. They are described in 3.3.10 Signals, p.149.

4.18 POSIX Memory Management

VxWorks provides the following POSIX memory management features in user-space (for real-time processes):

- Dynamic memory allocation—with the calloc(), malloc(), realloc(), and free() routines. See 5.3 Heap and Memory Partition Management, p.246.
- POSIX shared memory objects (the _POSIX.Shared.Memory.OBJECTS option). See 4.18.3 Shared Memory Objects, p.236.
- POSIX memory-mapped files (the _POSIX.MAPPED.FILES option). See 4.18.4 Memory Mapped Files, p.237.
- POSIX memory protection (the _POSIX.MEMORY.PROTECTION options). See 4.18.5 Memory Protection, p.238.
- POSIX memory locking (the _POSIX.MEMLOCK and _POSIX.MEMLOCK_RANGE options). See 4.18.6 Memory Locking, p.238.

In addition, VxWorks also supports anonymous memory mapping, which is not a POSIX standard, but an implementation-specific extension of it.

4.18.1 POSIX Memory Management APIs

The POSIX memory management APIs provided by VxWorks facilitate porting applications from other operating systems. Note, however, that the VxWorks implementation of POSIX memory management facilities is based on requirements for real-time operating systems, which include determinism, small-footprint, and scalability. Features that are commonly found in general-purpose operating systems, such as demand-paging and copy-on-write, are therefore not supported in VxWorks. This ensures deterministic memory access and execution time, but also means that system memory limits the amount of virtual memory that processes can map. In other words, memory mapped in processes are always
memory resident. Similarly, some APIs that are not relevant to real-time operating systems, such as those for memory locking are formally available to facilitate porting, but do not perform any function.

The `mmanLib` and `shmLib` routines are listed in Table 4-23.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mmap()</code></td>
<td>Establishes a mapping between a process’ address space and a file, or a shared memory object, or directly with system RAM. In VxWorks the MAP_FIXED flags parameter option is not supported.</td>
</tr>
<tr>
<td><code>munmap()</code></td>
<td>Un-maps pages of memory.</td>
</tr>
<tr>
<td><code>msync()</code></td>
<td>Synchronizes a mapped file with a physical storage.</td>
</tr>
<tr>
<td><code>mprotect()</code></td>
<td>Sets protection of memory mapping.</td>
</tr>
<tr>
<td><code>mlockall()</code></td>
<td>Locks all pages used by a process into memory. In VxWorks this routine does nothing.</td>
</tr>
<tr>
<td><code>munlockall()</code></td>
<td>Unlocks all pages used by a process. In VxWorks this routine does nothing.</td>
</tr>
<tr>
<td><code>mlock()</code></td>
<td>Locks specified pages into memory.</td>
</tr>
<tr>
<td><code>munlock()</code></td>
<td>Unlocks specified pages. In VxWorks this routine does nothing.</td>
</tr>
<tr>
<td><code>shm_open()</code></td>
<td>Opens a shared memory object.</td>
</tr>
<tr>
<td><code>shm_unlink()</code></td>
<td>Removes a shared memory object.</td>
</tr>
</tbody>
</table>
The restrictions on the VxWorks implementation of mmanLib are as follows:

- Mapping at fixed a address (MAP_FIXED) is not supported.
- The munmap(), mprotect(), msync(), mlock() routines only succeed for memory pages obtained with mmap().
- The msync() routine must be called explicitly to ensure that data for memory mapped files is synchronized with the media.
- If the file size changes after a mapping is created, the change is not reflected in the mapping. It is, therefore, important to make sure that files are not changed with the standard file operations—ftruncate(), write(), and so on—while mappings are in effect. For example, the second ftruncate() call in the following (abbreviated) example would have no effect on the mapping created with the mmap() call:

```c
fd = shm_open(...);
ftruncate(fd, size1); /* set size */
addr1 = mmap(..., fd); /* map it */
ftruncate(fd, size2); /* change size */
```

- Mappings that extend an existing shared mapping may not always succeed. This is because VxWorks uses a non-overlapped memory model that does not ensure adjacent virtual memory is available for a specific process.

### 4.18.2 Anonymous Memory Mapping

Anonymous memory-mapping is a VxWorks extension of the POSIX memory mapping APIs. It provides a simple method for a process to request (map) and release (unmap) additional pages of memory, without associating the mapping to a named file system object. This feature is always available when basic process (RTP) support included in VxWorks. Only private mappings are supported with the anonymous option.

The following example shows the use of mmap() with the anonymous flag, as well as mprotect(), and unmap().
#include <sys/mman.h>
#include <sys/sysctl.h>

/****************************************************************************
* main - User application entry function *
* This application illustrates the usage of the mmap(), mprotect(), and *
* munmap() API. It does not perform any other useful function. *
* EXIT STATUS: 0 if all calls succeeded, otherwise 1. */

int main ()
{
    int mib[3]; /* MIB array for sysctl() */
    size_t pgSize; /* variable to store page size */
    size_t sz = sizeof (pgSize); /* size of pgSize variable */
    size_t bufSize; /* size of buffer */
    char * pBuf; /* buffer */

    /* get "hw.mmu.pageSize" info */
    mib[0] = CTL_HW;
    mib[1] = HW_MMU;
    mib[2] = HW_MMU_PAGESIZE;
    if (sysctl (mib, 3, &pgSize, &sz, NULL, 0) != 0)
        exit (1);

    /* buffer size is 4 pages */
    bufSize = 4 * pgSize;

    /* request mapped memory for a buffer */
    pBuf = mmap (NULL, bufSize, (PROT_READ | PROT_WRITE),
                 (MAP_PRIVATE | MAP_ANON), MAP_ANON_FD, 0);

    /* check for error */
    if (pBuf == MAP_FAILED)
        exit (1);

    /* write protect the first page */
    if (mprotect (pBuf, pgSize, PROT_READ) != 0)
    {
        /*
        * no need to call unmap before exiting as all memory mapped for
        * a process is automatically released.
        */
        exit (1);
    }
}
/*
 * Unmap the buffer; the unmmap() has to be called for the entire buffer.
 * Note that unmapping before exit() is not necessary; it is shown here
 * only for illustration purpose.
 */

if (munmap (pBuf, bufSize) != 0)
    exit (1);

printf ("execution succeded\n");
exit (0);
}

For more information about the mmanLib routines, and an additional code example, see the VxWorks API reference for mmanLib.

### 4.18.3 Shared Memory Objects

VxWorks provides support for POSIX shared memory objects. With this type of mapping, the file descriptor used to call mmap() is obtained with shm_open(). Both shared and private mappings are supported. Support for this type of mapping is provided by two kernel components: a pseudo-file system called shmFs that provides the functionality for shm_open() and shm_unlink(), and the mmap() extension for mapped files. These facilities are provided in the INCLUDE_POSIX_SHM and INCLUDE_POSIX_MAPPED_FILES components, respectively.

The shared memory file system provides the name space for shared memory objects. It is a storage-less file system, which means that read, and write operations are not supported. The contents of shared memory objects can be managed exclusively by way of their memory mapped images.

Note that the functionality provided by shared memory objects functionality overlaps somewhat with the VxWorks native shared data region support. Table 4-24 summarizes their most important similarities and distinctions. For information about shared memory regions, see 2.7 Creating and Managing Shared Data Regions, p.60.
For more information, see the `shmLib` and `mmanLib` API references. For a code example, see the `shmLib` API.

### 4.18.4 Memory Mapped Files

With memory-mapped files, the file descriptor used to call `mmap()` is obtained by opening a regular file in a POSIX-compliant file system. Both shared and private mappings are supported. This type of mapping type is available when VxWorks is configured with the `INCLUDE_POSIX_MAPPED_FILES` component.

There is no automatic synchronization for memory mapped files, and there is no unified buffering for `mmap()` and the file system. This means that the application must use `msync()` to synchronize a mapped image with the file’s storage media.

---

**Table 4-24 Shared Memory and Shared Data Regions**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Shared Memory</th>
<th>Shared Data Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard, API portability</td>
<td>POSIX</td>
<td>proprietary</td>
</tr>
<tr>
<td>Name space</td>
<td>file system</td>
<td>named object</td>
</tr>
<tr>
<td>Identifier</td>
<td>file descriptor</td>
<td>object ID</td>
</tr>
<tr>
<td>Ability to map at fixed physical address</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Ability to map at fixed virtual address</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Available in the kernel</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Unit of Operation(^a)</td>
<td>MMU page</td>
<td>entire shared data region</td>
</tr>
<tr>
<td>Size management(^b)</td>
<td>dynamic</td>
<td>set at creation</td>
</tr>
</tbody>
</table>

\(^a\) *Unit of operation* means the portion of shared memory or a shared data region that can be operated on (mapped, unmapped, or protected). For shared memory the unit of operation is the MMU page (which means that shared memory can be partially mapped, unmapped, and protected on page basis). For shared data regions, all operations are on the entire region.

\(^b\) If the file size changes after a mapping is created, the change is not reflected in the mapping. It is, therefore, important to make sure that files are not changed with the standard file operations—`ftruncate()`, `write()`, and so on—while mappings are in effect.
The only exception to this rule is when memory is unmapped explicitly with `munmap()` or unmapped implicitly when the process exits. In that case, the synchronization is performed automatically during the unmapping process.

For more information, including a code example, see the mmanLib API reference.

4.18.5 Memory Protection

Memory protection of mappings established with `mmap()` can be changed using the `mprotect()` routine. It works with both anonymous memory mapped files and with shared memory objects.

For more information see the mprotect() API reference.

4.18.6 Memory Locking

In VxWorks, memory mappings are always memory resident. Demand paging and copy-on write are not performed. This ensures deterministic memory access for mapped files, but it also means that physical memory is always associated with mappings, until it is unmapped. The memory locking APIs provided with VxWorks only perform address space validation, and have no effect on the mappings.

For more information see the mmanLib API reference.
4.19 POSIX Trace

VxWorks provide support for the POSIX trace facility in user-space for real-time processes. The trace facility allows a process to select a set of trace event types, to initiate a trace stream of events as they occur, and to retrieve the record of trace events.

The trace facility can be used to for debugging application code during development, for monitoring deployed systems, and for performance measurement. When used for debugging, it can be used at run-time as well as for postmortem analysis. For run-time debugging, the trace facility should be used with the filtering mechanism to provide focus on specific information an to avoid swamping the trace stream and the system itself. For postmortem analysis, collection of comprehensive data is desirable, and tracing can be used at selected intervals.

A trace event is a recorded unit of the trace facility—it is the datum generated. Trace events include a trace event type identifier, a time-stamp, and other information. Standard POSIX trace events also have defined payloads; application-defined trace events may define a payload. Trace events are initiated either by a user application or by the trace system itself. The event structure is defined in installDir/vxworks-6.x/target/usr/h/base/b_struct_posix_trace_event_info.h.

A trace stream is a sequence of trace events, which are recorded in memory in a stream-buffer, and may be written to a file as well. The stream buffer is a ring-based buffer mechanism. Depending on the options selected when the stream is created, the buffer may stop logging when it is full, wrap (overwrite old data), or write the event data to a log file. When the trace facility is used without a trace log, the trace event data is lost when the tracing process is deleted or exits (note that the tracing process and the traced process may or may not be the same).

A trace log is a file to which a trace stream is written. A trace stream can be written to a log automatically (for example, when the posix_trace_shutdown() routine is called) or on demand. The trace log file format is not specified by the POSIX trace specification. The trace log is write-only while it is being created, and read-only thereafter.

Trace events are registered by name, and then recorded with a numeric identifier. Some events are defined in the POSIX standard. These have fixed numeric IDs and names. The symbolic values, and the textual names, are in the standard (for example POSIX_TRACE_START and posix_trace_start, POSIX_TRACE_OVERFLOW and posix_trace_overflow, and so on). These events types are included in any log you create. Other event type ID and name mappings can be created with the posix_trace_eventid_open() routine.
The `posix_trace_eventtypelist_getnext()` can be used to retrieve an ID, and then `posix_trace_eventid_get_name()` can be used to get the corresponding name.

There is a per-process limit on the number of events that are obtained, which may be obtained with the following call:

```
sysconf(THE_USER_EVENT_MAX);
```

Attempts to create event mappings beyond the limit return the special event ID `POSIX_TRACE_UNNAMED_USEREVENT`.

**Trace Operation**

The trace operation involves three logically distinct roles, each of which has its own APIs associated with it. These roles are the following:

- The **trace controller**, which governs the operation of recording trace events into the trace stream. This operation includes initializing the attributes of a trace stream, creating the stream, starting and stopping the stream, filtering the stream, and shutting the trace stream down, as well as other trace stream management and information retrieval functions.

- The **trace event generator**, which during execution of an instrumented application injects a trace record into a trace stream whenever a trace point is reached, as long as that type of trace has not been filtered out.

- The **trace analyzer**, which retrieves trace event data. It can either retrieve the traced events from an active trace stream or from a trace log file, extract stream attributes, look up event type names, iterate over event type identifiers, and so on.

While logically distinct, these roles can be performed by the same process, or by different processes. A process may trace itself or another process; and may have several traces active concurrently. A process may also open and read a trace log, provided the log was generated on the same CPU architecture as the process that is going to read it.

A trace event filter can be used to define specific types of trace events to collect, and to reduce the overall amount of data that is collected. All traces have an event filter, which records all events by default.

**Trace APIs**

The basic set of trace routines is the following:
More of the routines used for the controller, generator, and analyzer roles are described in Table 4-25.

Table 4-25  POSIX Trace Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_trace_attr_init( )</td>
<td>Initializes the trace stream attributes object.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_attr_getgenversion()</td>
<td>Gets the generation version information.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_attr_getinherited()</td>
<td>Gets the inheritance policy.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_attr_getmaxusereventsizes()</td>
<td>Calculates the maximum memory size required to store a single user trace event.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_create( )</td>
<td>Creates an active trace stream.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_clear( )</td>
<td>Re-initializes the trace stream.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_trid_eventid_open( )</td>
<td>Associates a user trace event name with a trace event type identifier</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_eventtypelist_getnext_id()</td>
<td>Returns the next event type.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_eventset_empty()</td>
<td>Excludes event types.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_set_filter()</td>
<td>Gets trace filter from a specified trace.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_start()</td>
<td>Starts a trace stream.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_get_attr()</td>
<td>Get a trace attributes.</td>
<td>Controller</td>
</tr>
<tr>
<td>posix_trace_eventid_open()</td>
<td>Associates and event name with an event-type.</td>
<td>Generator</td>
</tr>
<tr>
<td>posix_trace_event()</td>
<td>Adds event to trace stream.</td>
<td>Generator</td>
</tr>
<tr>
<td>posix_trace_attr_getgenversion()</td>
<td>Gets version information.</td>
<td>Analyzer</td>
</tr>
</tbody>
</table>
For more information about these and other APIs, see the `pxTraceLib` API reference entries.

To include the trace facility in your system, configure VxWorks with the `INCLUDE_POSIX_TRACE` component. This component is included automatically when the `BUNDLE_RTP_POSIX_PSE52` bundle is used.

### Trace Code and Record Example

The section provides an example of code that uses the trace facility and a trace record that it produces.

#### Trace Code Example

```c
LOCAL int eventShow
{
  trace_id_t trid,
  const struct posix_trace_event_info * pEvent,
  const void * pData,
  int dataSize
}
{
  int result;
  char eventName [TRACE_EVENT_NAME_MAX];
  char * truncationMsg;
  char * payloadData [32];
}
const char * dataString = (const char *)pData;

result = posix_trace_eventid_get_name (trid,
   pEvent->posix_event_id,
   eventName);
if (result != 0)
   sprintf (eventName, "error: %d", result);

vxTestMsg (V_GENERAL, "EventId:           %d (%s)",
   pEvent->posix_event_id,
   eventName);
vxTestMsg (V_GENERAL, "PID:               0x%x",  
   pEvent->posix_pid);
vxTestMsg (V_GENERAL, "Prog address:      0x%x",  
   pEvent->posix_prog_address);

switch (pEvent->posix_truncation_status)
{
   case POSIX_TRACE_NOT_TRUNCATED:
      truncationMsg = "POSIX_TRACE_NOT_TRUNCATED";
      break;
   case POSIX_TRACE_TRUNCATED_RECORD:
      truncationMsg = "POSIX_TRACE_TRUNCATED_RECORD";
      break;
   case POSIX_TRACE_TRUNCATED_READ:
      truncationMsg = "POSIX_TRACE_TRUNCATED_READ";
      break;
   default:
      truncationMsg = "Unknown";
}

vxTestMsg (V_GENERAL, "Truncation:        %d (%s)",
   pEvent->posix_truncation_status, truncationMsg);

vxTestMsg (V_GENERAL, "Time:              (%d, %d)",
   pEvent->posix_timestamp.tv_sec, pEvent->posix_timestamp.tv_nsec);
vxTestMsg (V_GENERAL, "ThreadId:          0x%x",  
   pEvent->posix_thread_id);

vxTestMsg (V_GENERAL, "Payload size:      %d", dataSize);

for (i=0; i < (dataSize < 16 ? dataSize : 16); i++)
{
   char ch = dataString [i];
   if (isspace(ch) || isgraph(ch))
      payloadData [i] = ch;
   else
      payloadData [i] = '.';
}
payloadData [i] = '\0';

vxTestMsg (V_GENERAL, "Payload data: %s", payloadData);
return (0);
Trace Event Record

EventId: 1025 (event1)
PID: 0x20001
Prog address: 0xa0002e90
Truncation: 0 (POSIX_TRACE_NOT_TRUNCATED)
Time: (16, 1367900)
ThreadId: 0xa0025410
Payload size: 0
Payload data:

EventId: 1025 (event1)
PID: 0x20001
Prog address: 0xa0002e90
Truncation: 0 (POSIX_TRACE_NOT_TRUNCATED)
Time: (16, 1368171)
ThreadId: 0xa0025410
Payload size: 0
Payload data:
5.1 Introduction

This chapter describes the memory management facilities available to applications that execute as real-time processes. Each process has its own heap, and can allocate and free buffers from its heap with the routines provided in the `memLib` and `memPartLib` libraries. See 5.3 Heap and Memory Partition Management, p.246.

In addition, run-time error detection facilities provide the ability to debug memory-related errors in application code. See 5.4 Memory Error Detection, p.248.

User applications can also make use of the following facilities for shared memory and memory mapping:

- Native VxWorks shared data regions (see 2.7 Creating and Managing Shared Data Regions, p.60).
- POSIX memory management facilities (see 4.18 POSIX Memory Management, p.232).
NOTE: This chapter provides information about facilities available for real-time processes. For information about the memory management facilities available in the VxWorks kernel, diagrams of memory for VxWorks configurations that includes processes, and instructions on how to configure VxWorks with process support—but without MMU support, see the VxWorks Kernel Programmer’s Guide: Memory Management.

5.2 VxWorks Component Requirements

When VxWorks is configured with real-time process support, it includes the basic memory management features required for process-based applications (see 2.2 Configuring VxWorks For Real-time Processes, p.8).

For the error detection features provided by heap and memory partition instrumentation, the INCLUDE_EDR_RTP_SHOW component is also required (5.4.2 Compiler Instrumentation, p.255).

5.3 Heap and Memory Partition Management

VxWorks provides support for heap and memory partition management in processes. By default, the heap is implemented as a memory partition within the process.

The heap is automatically created during the process initialization phase. The initial size of the heap, and the automatic incrementation size, are configurable with the environment variables HEAP_INITIAL_SIZE, HEAP_INCR_SIZE and HEAP_MAX_SIZE. These environment variables only have effect if they are set when the application is started. The application cannot change its own values. For more information on these environment variables, see the VxWorks API reference for memLib. For information about working with environment variables, see 2.3.5 Processes and Environment Variables, p.17, 2.5.1 Application Structure, p.19, and the API reference for rtpSpawn().
Memory partitions are contiguous areas of memory that are used for dynamic memory allocation by applications. Applications can create their own partitions and allocate and free memory from these partitions.

The heap and any partition created in a process are private to that process, which means that only that process is allowed to allocate memory to it, or free from it.

For more information, see the VxWorks API references for memPartLib and memLib.

Alternative Heap Manager

The VxWorks process heap implementation can be replaced by a custom version simply by linking the replacement library into the application (the replacement library must precede vxlib.a in the link order).

The memory used as heap can be obtained with either one of the following:

- A statically created array variable; for example:

```c
char heapMem[HEAP_SIZE];
```

This solution is simple, but it creates a fixed sized heap.

- Using the dynamic memory mapping function, `mmap()`. With `mmap()`, it is possible to implement automatic or non-automatic growth of the heap. However, it is important to keep in mind that subsequent calls to `mmap()` are not guaranteed to provide memory blocks that are adjacent. For more information about `mmap()` see 4.18.1 POSIX Memory Management APIs, p.232.

In case of applications that are dynamically linked with libc.so (which by default contains memLib.o), the default heap provided by memLib is automatically created. To avoid creation of the default heap, it is necessary to create a custom libc.so file that does not hold memLib.o. For information about creating a custom shared library that provides this functionality, please contact Wind River Support.

To ensure that process initialization code has access to the replacement heap manager early enough, the user-supplied heap manager must either:

- Initialize the heap automatically the very first time `malloc()` or any other heap routine is called.

- Have its initialization routine linked with the application, and declared as an automatic constructor using the _WRS_CONSTRUCTOR macro with an initialization order lower than 6 (for information about this macro, see 2.6.1 Library Initialization, p.28).
5.4 Memory Error Detection

Support for memory error detection is provided by two optional instrumentation libraries. The `memEdrLib` library performs error checks of operations in the user heap and memory partitions in a process. The Run-Time Error Checking (RTEC) feature of the Wind River Compiler can be used to check for additional errors, such as buffer overruns and underruns, static and automatic variable reference checks.

Errors detected by these facilities are reported with the logging facility of the error detection and reporting component; which must be included in the VxWorks configuration to provide this functionality.

For information about configuring VxWorks for error detection and reporting, as well as information about additional facilities useful for debugging software faults, see 8. Error Detection and Reporting.

NOTE: The memory error detection facilities described in this section are not included in any shared library provided by Wind River with this release. They can only be statically linked with application code.

5.4.1 Heap and Partition Memory Instrumentation

To supplement the error detection features built into `memLib` and `memPartLib` (such as valid block checking), the memory partition debugging library, `memEdrLib`, can be used to perform automatic, programmatic, and interactive error checks on `memLib` and `memPartLib` operations. This is performed by installing instrumentation hooks for `memPartLib`.

The instrumentation helps detect common programming errors such as double-freeing an allocated block, freeing or reallocating an invalid pointer, and memory leaks. In addition, with compiler-assisted code instrumentation, it helps detect bounds-check violations, buffer over-runs and under-runs, pointer references to free memory blocks, pointer references to automatic variables outside the scope of the variable, and so on.

**VxWorks Kernel Configuration**

A VxWorks kernel configured for process support (with the INCLUDE_RTP component) is sufficient for providing processes with heap instrumentation. The optional INCLUDE_MEM_EDR_RTP_SHOW component can be used to provide show routines for the heap and memory partition instrumentation. Note that the kernel’s heap instrumentation component (INCLUDE_MEM_EDR) is not required.
Linking

In order to enable heap and memory partition instrumentation of a process, the executable must be linked with the memEdrLib library support included. This can be accomplished by using the following linker option:

-\texttt{Wl,\-umemEdrEnable}

For example, the following makefile based on the provided user-side build environment can be used:

\begin{verbatim}
EXE = heapErr.vxe
OBJS = main.o
LD_EXEC_FLAGS += -Wl,\-umemEdrEnable
include $(WIND_USR)/make/rules.rtp
\end{verbatim}

Alternatively, adding the following lines to the application code can also be used:

\begin{verbatim}
extern int memEdrEnable;
memEdrEnable = TRUE;
\end{verbatim}

The location of these lines in the code is not important.

Environment Variables

When executing an application, the following environment variables may be set to override the defaults. The variables have to be set when the process is started. For information about working with environment variables, see 2.3.5 Processes and Environment Variables, p.17, 2.5.1 Application Structure, p.19, and the API reference for \texttt{rtpSpawn( )}.

**MEDR_EXTENDED_ENABLE**

Set to \texttt{TRUE} to enable saving trace information for each allocated block, but at the cost of increased memory used to store entries in the allocation database. Without this feature enabled the database entry for each allocated block is 32 bytes, with this feature enabled it is 64 bytes. Default setting is \texttt{FALSE}.

**MEDR_FILL_FREE_ENABLE**

Set to \texttt{TRUE} to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. Default setting is \texttt{FALSE}.

**MEDR_FREE_QUEUE_LEN**

Maximum allowed length of the free queue. When a memory block is freed by an application, instead of immediately returning it to the memory pool, it is kept in a queue. When the queue reaches the maximum length allowed, the blocks are returned to the memory pool in a FIFO order. Queuing is disabled when this parameter is 0. Default setting is 64.
MEDR_BLOCK_GUARD_ENABLE
Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, under-runs, and some heap memory corruption. Default setting is FALSE.

MEDR_POOL_SIZE
Set the size of the memory pool used to maintain the memory block database. Default setting in processes is 64 K. The database uses 32 bytes per memory block without extended information enabled, and 64 bytes per block with extended information enabled (call stack trace).

MEDR_SHOW_ENABLE
Enable heap instrumentation show support in the process. This is needed in addition to configuring VxWorks with the INCLUDE_MEM_EDR_RTP_SHOW component. When enabled, the kernel routines communicate with a dedicated task in the process with message queues. The default setting is FALSE.

Error Types
During execution, errors are automatically logged when the allocation, free, and re-allocation functions are called. The following error types are automatically identified and logged:

- Allocation returns a block address within an already allocated block from the same partition. This would indicate corruption in the partition data structures.
- Allocation returns a block address which is in the task’s stack space. This would indicate corruption in the partition data structures.
- Allocation returns a block address that is in the kernel’s static data section. This would indicate corruption in the partition data structures.
- Freeing a pointer which is in the task’s stack space.
- Freeing a memory that was already freed and is still in the free queue.
- Freeing memory which is in the kernel’s static data section.
- Freeing memory in a different partition than where it was allocated.
- Freeing a partial memory block.
- Freeing memory block with the guard zone corrupted, when the MEDR_BLOCK_GUARD_ENABLE environment variable is TRUE.
- Pattern in a memory block which is in the free queue has been corrupted, when the MEDR_FILL_FREE_ENABLE environment variable is TRUE.
### Shell Commands

The show routines and commands described in Table 5-1 are available for use with the shell’s C and command interpreters to display information.

<table>
<thead>
<tr>
<th>C Interpreter</th>
<th>Command Interpreter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edrShow()</td>
<td>edr show</td>
<td>Displays error records.</td>
</tr>
<tr>
<td>memEdrRtpPartShow()</td>
<td>mem rtp part list</td>
<td>Displays a summary of the instrumentation information for memory partitions located in a given process.</td>
</tr>
<tr>
<td>memEdrRtpBlockShow()</td>
<td>mem rtp block list</td>
<td>Displays information about allocated blocks in a given process. Blocks can be selected using a combination of various querying criteria: partition ID, block address, allocating task ID, block type.</td>
</tr>
<tr>
<td>memEdrRtpBlockMark()</td>
<td>mem rtp block mark</td>
<td>Marks or unmarks selected blocks allocated at the time of the call. The selection criteria may include partition ID and/or allocating task ID.</td>
</tr>
<tr>
<td>memEdrRtpBlockMark()</td>
<td>mem rtp block unmark</td>
<td>Can be used to monitor memory leaks by displaying information of unmarked blocks with memEdrRtpBlockShow() and mem rtp block list.</td>
</tr>
</tbody>
</table>
Code Example

The following application code can be used to generate various errors that can be monitored from the shell (line numbers are included for reference purposes). Its use is illustrated in Shell Session Example, p.252.

```c
#include <vxWorks.h>
#include <stdlib.h>
#include <taskLib.h>
int main ()
{
    char * pChar;
    taskSuspend(0);        /* stop here first */
    pChar = malloc (24);
    free (pChar + 2);      /* free partial block */
    free (pChar);
    free (pChar);          /* double-free block */
    pChar = malloc (32);   /* leaked memory */

    taskSuspend(0);       /* stop again to keep RTP alive */
}
```

Shell Session Example

First set up the environment variables in the shell task. These variables will be inherited by processes created with `rtpSp()`. The first environment variable enables trace information to be saved for each allocation, the second one enables the show command support inside the process.

```
-> putenv "MEDR_EXTENDED_ENABLE=TRUE"
value = 0 = 0x0
-> putenv "MEDR_SHOW_ENABLE=TRUE"
value = 0 = 0x0
```

Spawn the process using the executable produced from the example code:

```
-> rtp = rtpSp ("heapErr.vxe")
rtp = 0x223ced0: value = 36464240 = 0x22c6670
```

At this point, the initial process task (`iheapErr`), which is executing `main()`, is stopped at the first `taskSuspend()` call (line 9 of the source code). Now mark all allocated blocks in the process which resulted from the process initialization phase:

```
-> memEdrRtpBlockMark rtp
value = 27 = 0x1b
```

Next, clear all entries in the error log. This step is optional, and is used to limit the number of events displayed by the `edrShow()` command that will follow:

```
-> edrClear
value = 0 = 0x0
```
Resume the initial task `iheapErr` to continue execution of the application code:

```c
-> tr iheapErr
value = 0 = 0x0
```

After resuming the process will continue execution until the second `taskSuspend()` call (line 17). Now list all blocks in the process that are unmarked. These are blocks that have been allocated since `memEdrRtpBlockMark()` was called, but have not been freed. Such blocks are possible memory leaks:

```c
-> memEdrRtpBlockShow rtp, 0, 0, 0, 0, 1
```

<table>
<thead>
<tr>
<th>Addr</th>
<th>Type</th>
<th>Size</th>
<th>Part ID</th>
<th>Task ID</th>
<th>Task Name</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>30053970</td>
<td>alloc</td>
<td>32</td>
<td>30010698</td>
<td>22c8750</td>
<td>iheapErr</td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>malloc()</td>
<td>0x30004ae4()</td>
</tr>
</tbody>
</table>

```c
value = 0 = 0x0
```

Display the error log. The first error corresponds to line 12 in the test code, while the second error corresponds to line 14.

```c
-> edrShow
ERROR LOG
========
Log Size:          524288 bytes (128 pages)
Record Size:       4096 bytes
Max Records:       123
CPU Type:          0x5a
Errors Missed:     0 (old) + 0 (recent)
Error count:       2
Boot count:        4
Generation count:  6

==[1/2]==
Severity/Facility: NON-FATAL/RTP
Boot Cycle:        4
OS Version:        6.0.0
Time:              THU JAN 01 00:09:56 1970 (ticks = 35761)
Task:              "iheapErr" (0x022c8750)
RTP:               "heapErr.vxe" (0x022c6670)
RTP Address Space: 0x30000000 -> 0x30057000
freeing part of allocated memory block
PARTITION:         0x30010698
PTR=0x30053942
BLOCK: allocated at 0x30053940, 24 bytes

<<<<<<Traceback>>>>>>
0x300001b4 _start       +0x4c : main ()
0x300001e4 main         +0x2c : free ()
0x300007280 memPartFree +0x5c : 0x300004a10 ()
0x30004ac8 memEdrItemGet+0x6e8: 0x30004514 ()
0x30003cb8 memEdrErrorLog+0x138: saveRegs ()

253
Severity/Facility: NON-FATAL/RTP
Boot Cycle: 4
OS Version: 6.0.0
Time: THU JAN 01 00:09:56 1970 (ticks = 35761)
Task: "iheapErr" (0x022c8750)
RTP: "heapErr.vxe" (0x022c6670)
RTP Address Space: 0x30000000 -> 0x30057000

freeing memory in free list
PARTITION: 0x30010698
PTR=0x30053940
BLOCK: free block at 0x30053940, 24 bytes

<<<Traceback>>>>

0x300001b4 _start +0x4c : main ()
0x300001f4 main +0x3c : free ()
0x30007280 memPartFree +0x5c : 0x30004a10 ()
0x30004ac8 memEdrItemGet+0x6e8: 0x30004514 ()
0x30003cb8 memEdrErrorLog+0x138: saveRegs ()
value = 0 = 0x0

Finally, resume iheapErr again to allow it to complete and to be deleted:

-> tr iheapErr
value = 0 = 0x0
5.4.2 Compiler Instrumentation

Additional errors are detected if the application is compiled using the Run-Time Error Checking (RTEC) feature of the Wind River Compiler. The following flag should be used:

-\texttt{Xrtc=}\texttt{option}

\textbf{NOTE:} This feature is not available with the GNU compiler.

Code compiled with the -\texttt{Xrtc} flag is instrumented for run-time checks such as pointer reference check and pointer arithmetic validation, standard library parameter validation, and so on. These instrumentations are supported through the memory partition run-time error detection library. Table 5-2 lists the -\texttt{Xrtc} options that are supported.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>register and check static (global) variables</td>
</tr>
<tr>
<td>0x02</td>
<td>register and check automatic variables</td>
</tr>
<tr>
<td>0x08</td>
<td>pointer reference checks</td>
</tr>
<tr>
<td>0x10</td>
<td>pointer arithmetic checks</td>
</tr>
<tr>
<td>0x20</td>
<td>pointer increment/decrement checks</td>
</tr>
<tr>
<td>0x40</td>
<td>standard function checks; for example \texttt{memset()} and \texttt{bcopy()}</td>
</tr>
<tr>
<td>0x80</td>
<td>report source code filename and line number in error logs</td>
</tr>
</tbody>
</table>

The errors and warnings detected by the RTEC compile-in instrumentation are logged by the error detection and reporting facility (see 8. Error Detection and Reporting). The following error types are identified:

- Bounds-check violation for allocated memory blocks.
- Bounds-check violation of static (global) variables.
- Bounds-check violation of automatic variables.
- Reference to a block in the free queue.
- Reference to the free part of the task’s stack.
- De-referencing a NULL pointer.
Configuring VxWorks for RTEC Support

Support for this feature in the kernel is enabled by configuring VxWorks with the basic error detection and reporting facilities. See 8.2 Configuring Error Detection and Reporting Facilities, p.330.

Shell Commands

The compiler provided instrumentation automatically logs errors detected in applications using the error detection and reporting facility. For information about using shell commands with error logs, see 8.4 Displaying and Clearing Error Records, p.334.

Code Example

This application code generates various errors that can be recorded and displayed, if built with the Wind River Compiler and its -Xrtc option (line numbers are included for reference purposes). Its use is illustrated in Shell Session Example, p.256.

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

int main ()
{
    char   name[] = "very_long_name";
    char * pChar;
    int    state[] = { 0, 1, 2, 3 };;
    int    ix = 0;

    pChar = (char *) malloc (13);

    memcpy (pChar, name, strlen (name)); /* bounds check violation */
    /* of allocated block */

    for (ix = 0; ix < 4; ix++)
    state[ix] = state [ix + 1];        /* bounds check violation */
    /* of automatic variable */

    free (pChar);

    *pChar = '\0'; /* reference a free block */
}
```

Shell Session Example

In the following shell session example, the C interpreter is used to execute edrClear(), which clears the error log of any existing error records. Then the application is started with rtpSp(). Finally, the errors are displayed with edrShow().
First, clear the error log. This step is only performed to limit the number of events that are later displayed, when the events are listed:

```
-> edrClear
value = 0 = 0x0
```

Start the process using the executable created from the sample code listed above:

```
-> rtpSp "refErr.vxe"
value = 36283472 = 0x229a450
```

Next, list the error log. As shown below, three errors are detected by the compiler instrumentation:

```
-> edrShow
ERROR LOG
========
Log Size:         524288 bytes (128 pages)
Record Size:      4096 bytes
Max Records:      123
CPU Type:         0x5a
Errors Missed:    0 (old) + 0 (recent)
Error count:      3
Boot count:       4
Generation count: 8
```

The first one is caused by the code on line 13. A string of length 14 is copied into a allocated buffer of size 13:

```
==[1/3]================================================================
Severity/Facility:   NON-FATAL/RTP
Boot Cycle:          4
OS Version:          6.0.0
Time:                THU JAN 01 01:55:42 1970 (ticks = 416523)
Task:                "irefErr"  (0x0229c500)
RTP:                 "refErr.vxe" (0x0229a450)
RTP Address Space:   0x30000000 -> 0x30058000
Injection Point:     main.c:13
memory block bounds-check violation
PTR=0x30054940  OFFSET=0  SIZE=14
BLOCK: allocated at 0x30054940, 13 bytes
<<<<<Traceback>>>>>
0x300001b4 _start       +0x4c : main ()
0x300002ac main         +0xf4 : __rtc_chk_at ()
```

The second error refers to line 17. The local `state` array is referenced with index 4. Since the array has only four elements, the range of valid indexes is 0 to 3:

```
==[2/3]================================================================
Severity/Facility:   NON-FATAL/RTP
Boot Cycle:          4
OS Version:          6.0.0
```
memory block bounds-check violation
PTR=0x30022f34  OFFSET=16  SIZE=4
BLOCK: automatic at 0x30022f34, 16 bytes

The last error is caused by the code on line 21. A memory block that has been freed
is being modified:

pointer to free memory block
PTR=0x30054940  OFFSET=0  SIZE=1
BLOCK: free block at 0x30054940, 13 bytes

value = 0 = 0x0
6

I/O System

6.1 Introduction 260
6.2 Files, Devices, and Drivers 261
6.3 Basic I/O 263
6.4 Buffered I/O: stdio 274
6.5 Other Formatted I/O 276
6.6 Asynchronous Input/Output 277
6.7 Devices in VxWorks 280
6.1 Introduction

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

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

The VxWorks I/O system provides standard C libraries for both basic and buffered I/O. The basic I/O libraries are UNIX-compatible; the buffered I/O libraries are ANSI C-compatible.

The diagram in Figure 6-1 illustrates the relationships between the different elements of the VxWorks I/O system available to real-time processes (RTPs). All of these elements are discussed in this chapter.

Figure 6-1 Overview of the VxWorks I/O System for Processes
NOTE: This chapter provides information about facilities available for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the VxWorks Kernel Programmer’s Guide.

6.2 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 device drivers. In general, using the I/O system does not require any further understanding of the implementation of devices and drivers. Note, however, that the VxWorks I/O system gives drivers considerable flexibility in the way they handle each specific device. Drivers conform to the conventional user view presented here, but can differ in the specifics. See 6.7 Devices in VxWorks, p.280.

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.
6.2.1 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 indicates the file file1 on the DEV1: device.

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

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

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

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

NOTE: To be recognized by the virtual root file system, device names must begin with a single leading forward-slash, and must not contain any other slash characters. For more information, see 7.3 Virtual Root File System: VRFS, p. 295.

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

/usr

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

host:

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

File system devices using dosFs are often named with uppercase letters and digits followed by a colon. For example:
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 6-1.

Table 6-1 Basic I/O Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creat()</td>
<td>Creates a file.</td>
</tr>
<tr>
<td>remove()</td>
<td>Deletes a file.</td>
</tr>
<tr>
<td>open()</td>
<td>Opens a file (optionally, creates a file if it does not already exist.)</td>
</tr>
<tr>
<td>close()</td>
<td>Closes a file.</td>
</tr>
<tr>
<td>read()</td>
<td>Reads a previously created or opened file.</td>
</tr>
<tr>
<td>write()</td>
<td>Writes to a previously created or opened file.</td>
</tr>
<tr>
<td>ioctl()</td>
<td>Performs special control functions on files.</td>
</tr>
</tbody>
</table>
6.3.1 File Descriptors

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

File descriptors are not global. The kernel has its own set of file descriptors, and each process (RTP) has its own set. Tasks within the kernel, or within a specific process share file descriptors. The only instance in which file descriptors may be shared across these boundaries is when one process is a child of another process or of the kernel and it does not explicitly close a file using the descriptors it inherits from its parent. (Processes created by kernel tasks share only the spawning kernel task’s standard I/O file descriptors 0, 1 and 2.) For example:

- If task A and task B are running in process foo, and they each perform a `write()` on file descriptor 7, they will write to the same file (and device).
- If process bar is started independently of process foo (it is not foo’s child) and its tasks X and Y each perform a `write()` on file descriptor 7, they will be writing to a different file than tasks A and B in process foo.
- If process foobar is started by process foo (it is foo’s child) and its tasks M and N each perform a `write()` on file descriptor 7, they will be writing to the same file as tasks A and B in process foo. However, this is only true as long as the tasks do not close the file. If they close it, and subsequently open file descriptor 7, they will operate on a different file.

When a file is opened, a file descriptor is allocated and returned. When the file is closed, the file descriptor is deallocated.

**File Descriptor Table**

The size of the file descriptor table, which defines the maximum number of files that can be open simultaneously in a process, is inherited from the spawning environment. If the process is spawned by a kernel task, the size of the kernel file descriptor table is used for the initial size of the table for the new process.

The size of the file descriptor table for each process can be changed programmatically. The `rtpIoTableSizeGet()` routine reads the current size of the table, and the `rtpIoTableSizeSet()` routine changes it.

By default, file descriptors are reclaimed only when the file is closed for the last time. However, the `dup()` and `dup2()` routines can be used to duplicate a file descriptor. For more information, see 6.3.3 Standard I/O Redirection, p.265.

264
6.3.2 Standard Input, Standard Output, and Standard Error

Three file descriptors have special meanings:

- 0 is used for standard input (stdin).
- 1 is used for standard output (stdout).
- 2 is used for standard error output (stderr).

All real time processes read their standard input—like `getchar()`—from file descriptor 0. Similarly file descriptor 1 is used for standard output—like `printf()`.

And file descriptor 2 is used for outputting error messages. You can use these descriptors to manipulate the input and output for all tasks in a process at once by changing the files associated with these descriptors.

These standard file descriptors are used to make an application independent of its actual I/O assignments. If a process sends its output to standard output (where the file descriptor is 1), then its output can be redirected to any file of any device, without altering the application’s source code.

6.3.3 Standard I/O Redirection

If a process is spawned by a kernel task, the process inherits the standard I/O file descriptor assignments of the spawning kernel task. These may be the same as the global standard I/O file descriptors for the kernel, or they may be different, task-specific standard I/O file descriptors. (For more information about kernel standard I/O assignments, see the VxWorks Kernel Programmer’s Guide: I/O System.)

If a process is spawned by another process, it inherits the standard I/O file descriptor assignments of the spawning process.

After a process has been spawned, its standard I/O file descriptors can be changed to any file descriptor that it owns.

The POSIX `dup()` and `dup2()` routines are used for redirecting standard I/O to a different file and then restoring them, if necessary. (Note that this is a very different process from standard I/O redirection in the kernel).

The first routine is used to save the original file descriptors so they can be restored later. The second routine assigns a new descriptor for standard I/O, and can also be used to restore the original. Every duplicated file descriptor should be closed explicitly when it is no longer in use. The following example illustrates how the routines are used.
First use the `dup()` routine to duplicate and save the standard I/O file descriptors, as follows:

```c
/* Temporary fd variables */
int oldFd0;
int oldFd1;
int oldFd2;
int newFd;

/* Save the original standard file descriptor. */
oldFd0 = dup(0);
oldFd1 = dup(1);
oldFd2 = dup(2);
```

Then use `dup2()` to change the standard I/O files:

```c
/* Open new file for stdin/out/err */
newFd = open("newstandardoutputfile", O_RDWR, 0);

/* Set newFd to fd 0, 1, 2 */
dup2 (newFd, 0);
dup2 (newFd, 1);
dup2 (newFd, 2);
```

If the process’ standard I/O must be redirected again, the preceding step can be repeated with another new file descriptor.

If the original standard I/O file descriptors must be restored, the following procedure can be performed:

```c
/* When complete, restore the original standard IO */
dup2 (oldFd0, 0);
dup2 (oldFd1, 1);
dup2 (oldFd2, 2);

/* Close them after they are duplicated to fd 0, 1, 2 */
close (oldFd0);
close (oldFd1);
close (oldFd2);
```

This redirection only affect the process in which it is done. It does not affect the standard I/O of any other process or the kernel. Note, however, that any new processes spawned by this process inherit the current standard I/O file descriptors of the spawning process (whatever they may be) as their initial standard I/O setting.

For more information, see the VxWorks API references for `dup()` and `dup2()`.
6.3.4 Open and Close

Before I/O can be performed on a device, a file descriptor must be opened to the device by invoking the `open()` routine—or `creat()`, as discussed in the next section. The arguments to `open()` are the filename, the type of access, and the mode (file permissions):

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

For `open()` calls made in processes, the mode parameter is optional. The file-access options that can be used with the `flags` parameter to `open()` are listed in Table 6-2.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_RDONLY</td>
<td>Open for reading only.</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>Open for writing only.</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>Open for reading and writing.</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>Create a file if it does not already exist.</td>
</tr>
<tr>
<td>O_EXCL</td>
<td>Error on open if the file exists and O_CREAT is also set.</td>
</tr>
<tr>
<td>O_SYNC</td>
<td>Write on the file descriptor complete as defined by synchronized I/O file integrity completion.</td>
</tr>
<tr>
<td>O_DSYNC</td>
<td>Write on the file descriptor complete as defined by synchronized I/O data integrity completion.</td>
</tr>
<tr>
<td>O_RSYNC</td>
<td>Read on the file descriptor complete at the same sync level as O_DSYNC and O_SYNC flags.</td>
</tr>
<tr>
<td>O_APPEND</td>
<td>Set the file offset to the end of the file prior to each write, which guarantees that writes are made at the end of the file. It has no effect on devices other than the regular file system.</td>
</tr>
<tr>
<td>O_NONBLOCK</td>
<td>Non-blocking I/O.</td>
</tr>
<tr>
<td>O_NOCTTY</td>
<td>If the named file is a terminal device, don’t make it the controlling terminal for the process.</td>
</tr>
</tbody>
</table>
Note the following special cases with regard to use of the file access and mode (file permissions) parameters to `open( )`:

- In general, you can open only preexisting devices and files with `open( )`. However, with NFS network, dosFs, and HRFS devices, you can also create files with `open( )` by OR’ing `O_CREAT` with one of the other access flags.
- HRFS directories can be opened with the `open( )` routine, but only using the `O_RDONLY` flag.
- With both dosFs and NFS devices, you can use the `O_CREAT` flag to create a subdirectory by setting `mode` to `FSTAT_DIR`. Other uses of the mode parameter with dosFs devices are ignored.
- With an HRFS device you cannot use the `O_CREAT` flag and the `FSTAT_DIR` mode option to create a subdirectory. HRFS ignores the mode option and simply creates a regular file.
- The netDrv default file system does not support the `F_STAT_DIR` mode option or the `O_CREAT` flag.
- For NFS devices, the third parameter to `open( )` is normally used to specify the mode of the file. For example:

```c
myFd = open("fooFile", O_CREAT | O_RDWR, 0644);
```
- While HRFS supports setting the permission mode for a file, it is not used by the VxWorks operating system.
- Files can be opened with the `O_SYNC` flag, indicating that each write should be immediately written to the backing media. This flag is currently supported by the dosFs file system, and includes synchronizing the FAT and the directory entries.
- The `O_SYNC` flag has no effect with HRFS because file system is always synchronous. HRFS updates files as though the `O_SYNC` flag were set.

### Table 6-2  File Access Options  (cont'd)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_TRUNC</td>
<td>Open with truncation. If the file exists and is a regular file, and the file is successfully opened, its length is truncated to 0. It has no effect on devices other than the regular file system.</td>
</tr>
</tbody>
</table>

```
myFd = open("fooFile", O_CREAT | O_RDWR, 0644);
```
See the VxWorks file system API references for more information about the features that each file system supports.

The \texttt{open()} routine, if successful, returns a file descriptor. This file descriptor is then used in subsequent I/O calls to specify that file. The file descriptor is an identifier that is not task specific; that is, it is shared by all tasks within the memory space. Within a given process or the kernel, therefore, one task can open a file and any other task can then use the file descriptor. The file descriptor remains valid until \texttt{close()} is invoked with that file descriptor, as follows:

\begin{verbatim}
close (fd);
\end{verbatim}

At that point, I/O to the file is flushed (completely written out) and the file descriptor can no longer be used by any task within the process (or kernel). However, the same file descriptor number can again be assigned by the I/O system in any subsequent \texttt{open()}.

For processes, files descriptors are closed automatically only when a process terminates. It is, therefore, recommended that tasks running in processes explicitly close all file descriptors when they are no longer needed. As stated previously (6.3.1 File Descriptors, p.264), there is a limit to the number of files that can be open at one time. Note that a process owns the files, so that when a process is destroyed, its file descriptors are automatically closed.

### 6.3.5 Create and Remove

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

The \texttt{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 \texttt{creat()} are similar to those of \texttt{open()} except that the filename specifies the name of the new file rather than an existing one; the \texttt{creat()} routine returns a file descriptor identifying the new file.

\begin{verbatim}
fd = creat ("name", flag);
\end{verbatim}

Note that with the HRFS file system the \texttt{creat()} routine is POSIX-compliant, and the second parameter is used to specify file permissions; the file is opened in O_RDONLY mode.
With dosFs, however, the `creat()` routine is not POSIX-compliant and the second parameter is used for open mode flags.

The `remove()` routine deletes a named file on a file-system device:

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

Files should be closed before they are removed.

With non-file-system devices, the `creat()` routine performs the same function as `open()`. The `remove()` routine, however has no effect.

### 6.3.6 Read and Write

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

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

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

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

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

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

The `read()` and `write()` routines are POSIX-compliant.
6.3.7 File Truncation

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

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

If it succeeds in truncating the file, `ftruncate()` returns OK.

If the file descriptor refers to a device that cannot be truncated, `ftruncate()` returns ERROR, and sets `errno` to EINV AL.

If the size specified is larger than the actual size of the file, the result depends on the file system. For both dosFs and HRFS, the size of the file is extended to the specified size; however, for other file systems, `ftruncate()` returns ERROR, and sets `errno` to EINV AL (just as if the file descriptor referred to a device that cannot be truncated).

The `ftruncate()` routine is part of the POSIX 1003.1b standard. It is fully supported as such by the HRFS. The dosFs implementation is, however, only partially compliant: creation and modification times are not changed.

Also note that with HRFS the seek position is not modified by truncation, but with dosFs the seek position is set to the end of the file.

6.3.8 I/O Control

The `ioctl()` routine provides a flexible mechanism for performing I/O functions that are not performed by the other basic I/O calls. Examples include determining how many bytes are currently available for input, setting device-specific options, obtaining information about a file system, and positioning random-access files to specific byte positions.

The arguments to the `ioctl()` routine are the file descriptor, a code that identifies the control function requested, and an optional function-dependent argument:

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

For `ioctl()` calls made in processes, the `arg` parameter is optional. Both of the following are legitimate calls:

```c
fd = ioctl (fd, func, arg);
fd = ioctl (fd, func);
```
For example, the following call uses the `FIOBAUDRATE` function to set the baud rate of a `tty` device to 9600:

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

The discussion of specific devices in 6.7 Devices in VxWorks, p.280 summarizes the `ioctl()` functions available for each device. The `ioctl()` control codes are defined in `ioLib.h`. For more information, see the reference entries for specific device drivers or file systems.

The `ioctl()` routine is POSIX-compliant.

VxWorks also provides the `posix_devctl()` routine for special devices. For more information, see the API reference entry.

### 6.3.9 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 6-3.

Table 6-3  
**Select Macros**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD_ZERO</td>
<td>Zeroes all bits.</td>
</tr>
<tr>
<td>FD_SET</td>
<td>Sets the bit corresponding to a specified file descriptor.</td>
</tr>
<tr>
<td>FD_CLR</td>
<td>Clears a specified bit.</td>
</tr>
<tr>
<td>FD_ISSET</td>
<td>Returns non-zero if the specified bit is set; otherwise returns 0.</td>
</tr>
</tbody>
</table>

Applications can use `select()` with any character I/O devices that provide support for this facility (for example, pipes, serial devices, and sockets).
6.3.10 POSIX File System Routines

The POSIX fsPxLib library provides I/O and file system routines for various file manipulations. These routines are described in Table 6-4.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlink()</td>
<td>Unlink a file.</td>
</tr>
<tr>
<td>link()</td>
<td>Link a file.</td>
</tr>
<tr>
<td>fsync()</td>
<td>Synchronize a file.</td>
</tr>
<tr>
<td>fdatasync()</td>
<td>Synchronize a file data.</td>
</tr>
<tr>
<td>rename()</td>
<td>Change the name of a file.</td>
</tr>
<tr>
<td>fpathconf()</td>
<td>Determine the current value of a configurable limit.</td>
</tr>
<tr>
<td>pathconf()</td>
<td>Determine the current value of a configurable limit.</td>
</tr>
<tr>
<td>access()</td>
<td>Determine accessibility of a file.</td>
</tr>
<tr>
<td>chmod()</td>
<td>Change the permission mode of a file.</td>
</tr>
<tr>
<td>fcntl()</td>
<td>Perform control functions over open files.</td>
</tr>
</tbody>
</table>

For more information, see the API references for fsPxLib and ioLib.

6.4 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.
6.4.1 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:

```c
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`):

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

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

A file descriptor that is already open can be associated subsequently with a `FILE` buffer by calling `fdopen()`:

```c
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.
The FILE buffer is deallocated when fclose() is called.

6.4.2 Standard Input, Standard Output, and Standard Error

As discussed in 6.3 Basic I/O, p. 263, there are three special file descriptors (0, 1, and 2) reserved for standard input, standard output, and standard error. Three corresponding stdio FILE buffers are automatically created when a task uses the standard file descriptors, stdin, stdout, and stderr, to do buffered I/O to the standard file descriptors. Each task using the standard I/O file descriptors has its own stdio FILE buffers. The FILE buffers are deallocated when the task exits.

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

6.5 Other Formatted I/O

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

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

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

Include AIO in your VxWorks configuration with the INCLUDE_POSIX_AIO and INCLUDE_POSIX_AIO_SYSDRV components. The second configuration constant enables the auxiliary AIO system driver, required for asynchronous I/O on all current VxWorks devices.

NOTE: The asynchronous I/O facilities are not included in any RTP shared library provided by Wind River for use with this release. They can only be statically linked with application code. For information about creating a custom shared library that provides this functionality, please contact Wind River Support.

6.6.1 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 6-5.
6.6.2 AIO Control Block

Each of the AIO calls takes an AIO control block (aiocb) as an argument. The calling routine must allocate space for the aiocb, and this space must remain available for the duration of the AIO operation. (Thus the aiocb must not be created on the task’s stack unless the calling routine will not return until after the AIO operation is complete and aio_return() has been called.) Each aiocb describes a single AIO operation. Therefore, simultaneous asynchronous I/O operations using the same aiocb are not valid and produce undefined results.

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

aiocb_fildes
   The file descriptor for I/O.

aiocb_offset
   The offset from the beginning of the file.

aiocb_buf
   The address of the buffer from/to which AIO is requested.

aiocb_nbytes
   The number of bytes to read or write.

aiocb_reqprio
   The priority reduction for this AIO request.
aio_sigevent
The signal to return on completion of an operation (optional).

aio_lio_opcode
An operation to be performed by a lio_listio() call.

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

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

⚠️ **CAUTION:** The aiocb structure and the data buffers referenced by it are used by the system to perform the AIO request. Therefore, once the aiocb has been submitted to the system, the application must not modify the aiocb structure until after a subsequent call to aio_return(). The aio_return() call retrieves the previously submitted AIO data structures from the system. After the aio_return() call, the calling application can modify the aiocb, free the memory it occupies, or reuse it for another AIO call. If space for the aiocb is allocated from the stack, the task should not be deleted (or complete running) until the aiocb has been retrieved from the system with an aio_return() call.

### 6.6.3 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(). To force all I/O operations to the synchronized I/O completion state, use aio_fsync().
Alternatives for Testing AIO Completion

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

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

6.7 Devices in VxWorks

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


6.7.1 Serial I/O Devices: Terminal and Pseudo-Terminal Devices

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

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

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

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

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

For more information about I/O control functions, see *VxWorks Kernel Programmer’s Guide: I/O System*.

Table 6-6 is a summary of the available options. The listed names are defined in the header file `ioLib.h`. For more detailed information, see the API reference entry for `tyLib`.

**Table 6-6  Tty Options**

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT_LINE</td>
<td>Selects line mode. (See <em>Raw Mode and Line Mode</em>, p.282.)</td>
</tr>
<tr>
<td>OPT_ECHO</td>
<td>Echoes input characters to the output of the same channel.</td>
</tr>
</tbody>
</table>
| 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.                      |
| OPT_MON_TRAP  | Enables the special `ROM monitor trap` character, `CTRL+X` by default.      |
| OPT_ABORT     | Enables the special kernel shell abort character, `CTRL+C` by default.      |
|               | (Only useful if the kernel shell is configured into the system)            |
| OPT_TERMINAL  | Sets all of the above option bits.                                         |
| OPT_RAW       | Sets none of the above option bits.                                         |
**Raw Mode and Line Mode**

A *tty* device operates in one of two modes: *raw mode* (unbuffered) or *line mode*. Raw mode is the default. Line mode is selected by the **OPT_LINE** bit of the device option word (see *tty Options*, p.281).

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.282.

**tty Special Characters**

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

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

- The line-delete character, by default **CTRL+U**, deletes all the characters of the current line.

- The end-of-file (EOF) character, by default **CTRL+D**, causes the current line to become available in the input ring without a **NEWLINE** and without entering the EOF character itself. Thus if the EOF character is the first character typed on a line, reading that line returns a zero byte count, which is the usual indication of end-of-file.

The following characters have special effects if the *tty* device is operating with the corresponding option bit set:

- The software flow control characters **CTRL+Q** and **CTRL+S** (XON and XOFF). Receipt of a **CTRL+S** input character suspends output to that channel. Subsequent receipt of a **CTRL+Q** resumes the output. Conversely, when the VxWorks input buffer is almost full, a **CTRL+S** is output to signal the other side
to suspend transmission. When the input buffer is empty enough, a CTRL+Q is output to signal the other side to resume transmission. The software flow control characters are enabled by OPT_TANDEM.

- The ROM monitor trap character, by default CTRL+X. This character traps to the ROM-resident monitor program. Note that this is drastic. All normal VxWorks functioning is suspended, and the computer system is controlled entirely by the monitor. Depending on the particular monitor, it may or may not be possible to restart VxWorks from the point of interruption.1 The monitor trap character is enabled by OPT_MON_TRAP.

- The special kernel shell abort character, by default CTRL+C. This character restarts the kernel shell if it gets stuck in an unfriendly routine, such as one that has taken an unavailable semaphore or is caught in an infinite loop. The kernel shell abort character is enabled by OPT_ABORT.

The characters for most of these functions can be changed using the ttyLib routines shown in Table 6-7.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL+H</td>
<td>backspace (character delete)</td>
<td>tyBackspaceSet( )</td>
</tr>
<tr>
<td>CTRL+U</td>
<td>line delete</td>
<td>tyDeleteLineSet( )</td>
</tr>
<tr>
<td>CTRL+D</td>
<td>EOF (end of file)</td>
<td>tyEOFSet( )</td>
</tr>
<tr>
<td>CTRL+C</td>
<td>kernel shell abort</td>
<td>tyAbortSet( )</td>
</tr>
<tr>
<td>CTRL+X</td>
<td>trap to boot ROMs</td>
<td>tyMonitorTrapSet( )</td>
</tr>
<tr>
<td>CTRL+S</td>
<td>output suspend</td>
<td>N/A</td>
</tr>
<tr>
<td>CTRL+Q</td>
<td>output resume</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.7.2 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

1. It will not be possible to restart VxWorks if un-handled external interrupts occur during the boot countdown.
other tasks. Pipe devices are managed by pipeDrv and use the kernel message queue facility to bear the actual message traffic.

Named pipes can be created in processes. However, unless they are specifically deleted by the application they will persist beyond the life of the process in which they were created. Applications should allow for the possibility that the named pipe already exists, from a previous invocation, when the application is started.

Creating Pipes

Pipes are created by calling the pipe create routine:

```c
status = pipeDevCreate ("/pipe/name", maxMsgs, maxLength);
```

The new pipe can have at most maxMsgs messages queued at a time. Tasks that write to a pipe that already has the maximum number of messages queued are blocked until a message is dequeued. Each message in the pipe can be at most maxLength bytes long; attempts to write longer messages result in an error.

I/O Control Functions

Pipe devices respond to the ioctl() functions summarized in Table 6-8. 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>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIOFLUSH</td>
<td>Discards all messages in the pipe.</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>Gets the pipe name of the file descriptor.</td>
</tr>
<tr>
<td>FIONMSGS</td>
<td>Gets the number of messages remaining in the pipe.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>Gets the size in bytes of the first message in the pipe.</td>
</tr>
</tbody>
</table>

6.7.3 Pseudo I/O Device

The memDrv device allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. The device provides a high-level means for reading and writing bytes in
absolute memory locations through I/O calls. It is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

For information about implementing `memDrv`, see the *VxWorks Kernel Programmer’s Guide: I/O System*.

For information about creating a RAM disk, which provides support for file systems, see the *VxWorks Kernel Programmer’s Guide: I/O System*.

### I/O Control Functions

The memory device responds to the `ioctl()` functions summarized in Table 6-9. The functions listed are defined in the header file `ioLib.h`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIOSEEK</td>
<td>Sets the current byte offset in the file.</td>
</tr>
<tr>
<td>FLOWHERE</td>
<td>Returns the current byte position in the file.</td>
</tr>
</tbody>
</table>

For more information, see the reference entries for `memDrv`, `ioLib`, and `ioctl()`.

### 6.7.4 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.

Using NFS devices, you can create, open, and access remote files exactly as though they were on a file system on a local disk. This is called *network transparency*.

For detailed information about the VxWorks implementation of NFS, see the *VxWorks Kernel Programmer’s Guide: Network File System*. 
I/O Control Functions for NFS Clients

NFS client devices respond to the ioctl() functions summarized in Table 6-10. The functions listed are defined in ioLib.h. For more information, see the reference entries for nfsDrv, ioLib, and ioctl().

Table 6-10  I/O Control Functions Supported by nfsDrv

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIOFSTATGET</td>
<td>Gets file status information (directory entry data).</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>Gets the filename of the file descriptor.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>Gets the number of unread bytes in the file.</td>
</tr>
<tr>
<td>FIOREADDR</td>
<td>Reads the next directory entry.</td>
</tr>
<tr>
<td>FIOSEEK</td>
<td>Sets the current byte offset in the file.</td>
</tr>
<tr>
<td>FIOSYNC</td>
<td>Flushes data to a remote NFS file.</td>
</tr>
<tr>
<td>FLOWHERE</td>
<td>Returns the current byte position in the file.</td>
</tr>
</tbody>
</table>

6.7.5 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, which is included in the Wind River Network Stack. Using this driver, you can open, read, write, and close files located on remote systems without needing to manage the details of the underlying protocol used to effect the transfer of information. (For more information, see the Wind River Network Stack for VxWorks 6 Programmer’s Guide.)

When a remote file is opened using RSH or FTP, the entire file is copied into local memory. As a result, the largest file that can be opened is restricted by the available memory. Read and write operations are performed on the memory-resident copy of the file. When closed, the file is copied back to the original remote file if it was modified.

In general, NFS devices are preferable to RSH and FTP devices for performance and flexibility, because NFS does not copy the entire file into local memory. However, NFS is not supported by all host systems.
I/O Control Functions

RSH and FTP devices respond to the same ioctl() functions as NFS devices except for FIOSYNC and FIOREADDIR. The functions are defined in the header file ioLib.h. For more information, see the API reference entries for netDrv and ioctl().

6.7.6 Sockets

In VxWorks, the underlying basis of network communications is sockets. A socket is an endpoint for communication between tasks; data is sent from one socket to another. Sockets are not created or opened using the standard I/O functions. Instead, they are created by calling socket(), and connected and accessed using other routines in sockLib. However, after a stream socket (using TCP) is created and connected, it can be accessed as a standard I/O device, using read(), write(), ioctl(), and close(). The value returned by socket() as the socket handle is in fact an I/O system file descriptor.

VxWorks socket routines are source-compatible with the BSD 4.4 UNIX socket functions and the Windows Sockets (Winsock 1.1) networking standard. Use of these routines is discussed in Wind River Network Stack for VxWorks 6 Programmer’s Guide.

6.7.7 Transaction-Based Reliable File System Facility: TRFS

The transaction-based reliable file system (TRFS) facility provides a fault-tolerant file system I/O layer for the dosFs file system. It is provided with the INCLUDE_XBD_TRANS component.

TRFS provides both file system consistency and fast recovery for the dosFs file system—DOS-compatible file systems are themselves neither reliable nor transaction-based. It is designed to operate with XBD-compliant device drivers for hard disks, floppy disks, compact flash media, TrueFFS flash devices, and so on. It can also be used with the XBD wrapper component for device drivers that are not XBD-compliant.
TRFS provides reliability in resistance to sudden power loss: files and data that are already written to media are unaffected, they will not be deleted or corrupted because data is always written either in its entirety or not at all.

TRFS provides additional guarantees in its transactional feature: data is always maintained intact up to a given commit transaction. User applications set transaction points on the file system. If there is an unexpected failure of the system, the file system is returned to the state it was in at the last transaction point. That is, if data has changed on the media after a commit transaction but prior to a power loss, it is automatically restored to the its state at the last commit transaction to further ensure data integrity. On mounting the file system, TRFS detects any failure and rolls back data to the last secure transaction.

Unlike some facilities that provide data integrity on a file-by-file basis, TRFS protects the medium as a whole. It is transactional for a file system, which means that setting transaction points will commit all files, not just the one used to set the transaction point.

**NOTE:** While TRFS is a I/O layer added to dosFs, it uses a modified on-media format that is not compatible with other FAT-based file systems, including Microsoft Windows and the VxWorks dosFs file system without the TRFS layer. It should not, therefore, be used when compatibility with other systems is a requirement.

For information about dosFs, see *7.5 MS-DOS-Compatible File System: dosFs*, p.304.

**Configuring VxWorks With TRFS**

Configure VxWorks with the `INCLUDE_XBD_TRANS` component to provide TRFS functionality for your dosFs file system.

**Automatic Instantiation of TRFS**

TRFS is automatically detected and instantiated if the media has already been formatted for use with TRFS, in a manner very similar to the instantiation of the dosFs or HRFS file system. The primary difference is that when TRFS is detected by the file system monitor, it calls the TRFS creation function, and the creation function then creates another XBD instance and generates an insertion event for it. The monitor then detects the new XBD and begins probing. In this case, however, the monitor does not examine the media directly—all commands are routed through TRFS, which performs the appropriate translations. If a file system is
detected, such as dosFs, the dosFs creation function is called by the monitor and
dosFs is instantiated. If not, rawfs is instantiated.

For information about how file systems are automatically instantiated, see the

Using TRFS in Applications

Once TRFS and dosFs are created, the dosFs file system may be used with the
ordinary file creation and manipulation commands. No changes to the file system
become permanent, however, until TRFS is used to commit them.

It is important to note that the entire dosFs file system—and not individual files—
are committed. The entire disk state must therefore be consistent before executing
a commit; that is, there must not be a file system operation in progress (by another
task, for example) when the file system is committed. If multiple tasks update the
file system, care must be taken to ensure the file data is in a known state before
setting a transaction point.

To commit a file system from a process, call:

```
ioctl(fd, CBIO_TRANS_COMMIT, 0);
```

where `fd` is any file descriptor that is open.

TRFS Code Example

The following code example illustrates setting a transaction point.

```c
void transTrfs
(
    void
)
{
    int fd;
    /* This assumes a TRFS with DosFs on "/trfs" */
    fd = open("/trfs/test.1", O_RDWR | O_CREAT, 0);
    ... /* Perform file operations here */
    ioctl(fd, CBIO_TRANS_COMMIT, 0);
    ... /* Perform more file operations here */
    ioctl(fd, CBIO_TRANS_COMMIT, 0);
    close(fd);
}
```
7

Local File Systems

7.1 Introduction 292
7.2 File System Monitor 295
7.3 Virtual Root File System: VRFS 295
7.4 Highly Reliable File System: HRFS 297
7.5 MS-DOS-Compatible File System: dosFs 304
7.6 Raw File System: rawFs 316
7.7 CD-ROM File System: cdromFs 319
7.8 Read-Only Memory File System: ROMFS 323
7.9 Target Server File System: TSFS 325
7.1 Introduction

VxWorks provides a variety of file systems that are suitable for different types of applications. The file systems can be used simultaneously, and in most cases in multiple instances, for a single VxWorks system.

Most VxWorks file systems rely on the extended block device (XBD) facility for a standard I/O interface between the file system and device drivers. This standard interface allows you to write your own file system for VxWorks, and freely mix file systems and device drivers.

File systems used for removable devices make use of the file system monitor for automatic detection of device insertion and instantiation of the appropriate file system on the device.

The relationship between applications, file systems, I/O facilities, device drivers and hardware devices is illustrated in Figure 7-1. Note that this illustration is relevant for the HRFS, dosFs, rawFs, and cdromFs file systems. The dotted line indicates the elements that must be configured and instantiated to create a specific, functional run-time file system.
For information about the XBD facility, see the *VxWorks Kernel Programmer’s Guide: I/O System*.
This chapter discusses the following VxWorks file systems and how they are used:

- **VRFS**
  A virtual root file system for use with applications that require a POSIX root file system. The VRFS is simply a root directory from which other file systems and devices can be accessed. See 7.3 Virtual Root File System: VRFS, p.295.

- **HRFS**
  A POSIX-compliant transactional file system designed for real-time use of block devices (disks). Can be used on flash memory in conjunction with TrueFFS and the XBD block wrapper component. See 7.4 Highly Reliable File System: HRFS, p.297.

- **dosFs**
  An MS-DOS compatible file system designed for real-time use of block devices. Can be used with flash memory in conjunction with the TrueFFS and the XBD block wrapper component. Can also be used with the transaction-based reliable file system (TRFS) facility. See 7.5 MS-DOS-Compatible File System: dosFs, p.304.

- **rawFS**
  Provides a simple raw file system that treats an entire disk as a single large file. See 7.6 Raw File System: rawFs, p.316.

- **cdromFs**
  Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system. See 7.7 CD-ROM File System: cdromFs, p.319.

- **ROMFS**
  Designed for bundling applications and other files with a VxWorks system image. No storage media is required beyond that used for the VxWorks boot image. See 7.8 Read-Only Memory File System: ROMFS, p.323.

- **TSFS**
  Uses the host target server to provide the target with access to files on the host system. See 7.9 Target Server File System: TSFS, p.325.

For information about the file system monitor, see the VxWorks Kernel Programmer’s Guide: Local File Systems. For information about the XBD facility, see the VxWorks Kernel Programmer’s Guide: I/O System.
7 Local File Systems

7.2 File System Monitor

File Systems and Flash Memory

VxWorks can be configured with file-system support for flash memory devices using TrueFFS and the HRFS or dosFs file system. For more information, see 7.5 MS-DOS-Compatible File System: dosFs, p.304 and the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS.

NOTE: This chapter provides information about facilities available for real-time processes. For information about creating file systems, and file system facilities available in the kernel, see the VxWorks Kernel Programmer’s Guide: Local File Systems.

7.2 File System Monitor

The file system monitor provides for automatic detection of device insertion, and instantiation of the appropriate file system on the device. The monitor is required for all file systems that are used with the extended block device (XBD) facility. It is provided with the INCLUDE_FS_MONITOR component.

The file systems that require both the XBD and the file system monitor components are HRFS, dosFs, rawFs, and cdromFs.

For detailed information about how the file system monitor works, see the VxWorks Kernel Programmer’s Guide: Local File Systems.

7.3 Virtual Root File System: VRFS

VxWorks provides a virtual root file system (VRFS) for use with applications that require a POSIX root file system. The VRFS is simply a “/” or root directory from which other file systems and devices can be accessed. VRFS is not a true file system, as files and directories cannot be created with the sorts of commands that are ordinarily associated with file systems, and it is read-only.

Only devices whose names begin with a single leading forward slash—and which do not contain any other forward slash characters—are recognized by the VRFS.
To include the VRFS in VxWorks, configure the kernel with the `INCLUDE_VRFS` component. The VRFS is created and mounted automatically if the component is included in VxWorks.

This shell session illustrates the relationship between device names and access to devices and file systems with the VRFS.

```bash
-> dev
drv name
 0 /null
 1 /tyCo/0
 1 /tyCo/1
 2 /aioPipe/0x1817040
 6 /romfs
 7 /
 9 yow-build02-lx:
10 /vio
11 /shm
12 /ram0
value = 25 = 0x19

-> cd /
value = 0 = 0x0
-> ll
?---------- 0 0       0                0 Jan  1 00:00 null
drwxrwxr-x 0 15179   100             20 Jan 23  2098 romfs/
?---------- 0 0       0                0 Jan  1 00:00 vio
drwxrwxrwx 1 0       0                0 Jan  1 00:00 shm/
drwxrwxrwx 1 0       0                2048 Jan  1 00:00 ram0/
value = 0 = 0x0
```

Note that `/tyCo/0`, `/tyCo/1`, `/aioPipe/0x1817040` and `yow-build02-lx` do not show up in the directory listing of the root directory as they do not follow the naming convention required by the VRFS. The first three include forward slashes in the body of the device name and the fourth does not have a leading forward slash in its name.

Also note that the listings of file systems have a trailing forward slash character. Other devices do not, and they have a question mark in the permissions (or attributes) column of the listing because they do not have recognizable file permissions.

⚠️ CAUTION: VRFS alters the behavior of other file systems because it provides a root directory on VxWorks. Changing directory to an absolute path on a host file system will not work when VRFS is installed without preceding the absolute path with the VxWorks device name. For example, if the current working directory is `hostname`, changing directory to `/home/panloki` will not work—it must be named `hostname:/home/panloki`. 
7.4 Highly Reliable File System: HRFS

The Highly Reliable File System (HRFS) is a transactional file system for real-time systems. The primary features of the file system are:

- Fault tolerance. The file system is never in an inconsistent state, and is therefore able to recover quickly from unexpected loses of power.
- Transactional operations on a file basis, rather than the whole disk.
- Hierarchical file and directory system, allowing for efficient organization of files on a volume.
- Compatibility with a widely available storage devices.
- POSIX compliance.

For more information about the HRFS libraries see the VxWorks API references for hrfsFormatLib, hrFsLib, and hrfsChkDskLib.

HRFS and Flash Memory

For information about using HRFS with flash memory, see the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS.

7.4.1 Configuring VxWorks for HRFS

To include HRFS support in VxWorks, configure the kernel with the appropriate required and optional components.

Required Components

The components required for HRFS are the following:

- Either the INCLUDE_HRFS or the INCLUDE_HRFS_READONLY component is required. As its name indicates, the latter is a read-only version of the main HRFS component. The libraries it provides are smaller as it provides no facilities for disk modifications.
- The appropriate component for your block device; for example, INCLUDE_ATA.
- If you are using a device driver that is not designed for use with the XBD facility, you must use the INCLUDE_XBD_BLK_DEV wrapper component in addition to INCLUDE_XBD. See the VxWorks Kernel Programmer’s Guide: I/O System for more information.
Optional HRFS Components

The INCLUDE_HRFS_FORMAT (formatter), INCLUDE_HRFS_CHKDSK (consistency checker), and INCLUDE_HRFS_ACCESS_TIMESTAMP (access file stamp) components are optional. Note that the INCLUDE_HRFS_ACCESS_TIMESTAMP component is included in the BUNDLE_RTP_POSIX_PSE52 component bundle.

Optional XBD Components

The INCLUDE_XBD_PART_LIB (disk partitioning) and INCLUDE_XBD_RAMDRV (RAM disk) components are optional.

For information about the XBD facility, see the VxWorks Kernel Programmer's Guide: I/O System.

7.4.2 HRFS and POSIX PSE52

If an application requires a PSE52 file system, then VxWorks must be configured with the following components:

- The INCLUDE_HRFS component for the highly reliable file system.
- The INCLUDE_HRFS_ACCESS_TIMESTAMP component for updating file and directory access timestamps on an HRFS formatted volume must be included. This component is part of the BUNDLE_RTP_POSIX_PSE52 bundle.

⚠️ WARNING: The use of the INCLUDE_HRFS_ACCESS_TIMESTAMP component can cause significant degradation in file system performance. For more information, see 7.4.5 File Access Time Stamps, p.299.

- The appropriate device driver component (for example INCLUDE_ATA or INCLUDE_XBD_RAMDRV) and INCLUDE_XBD_BLK_DEV if the device driver requires it. See the VxWorks Kernel Programmer's Guide: I/O System.
- The INCLUDE_VRFS component for the VRFS file system, which is included in the BUNDLE_RTP_POSIX_PSE52 component bundle.

In addition, a /tmp directory must be created at run-time. It must appear on the VRFS file system, and must be formatted as an HRFS file system. The following examples illustrate creation of a /tmp RAM disk device and formatting the directory as an HRFS file system:

```
xbdRamDiskDevCreate (512, 0x10000, 0, "/tmp")
hhrsDiskFormat "/tmp", 1000.
```
7.3 Virtual Root File System: VRFS, p.295 and 4.2.1 POSIX PSE52 Support, p.166.

7.4.3 Creating an HRFS File System

For information about creating an HRFS file system, see VxWorks Kernel Programmer’s Guide: Local File Systems.

7.4.4 Transactional Operations

HRFS is a transactional file system. Transactions are performed automatically on a file or directory basis when they are changed. (The TRFS facility—which can be used with dosFs—performs transactions on the entire disk; see 6.7.7 Transaction-Based Reliable File System Facility: TRFS, p.287.)

Transactions are committed to disk automatically when a file or directory is modified or deleted. That is, upon successful completion of a function that modifies the disk, the modifications are committed. There is no need for explicit action on the part of an application to commit a change. For example, the following routines cause modifications to disk:

- write()
- remove()
- delete()
- mkdir()
- rmdir()
- link()
- unlink()
- truncate()
- truncated()
- ioctl() when used with a control function that requires modifying the disk.

7.4.5 File Access Time Stamps

Access time stamps can be enabled by configuring VxWorks with the INCLUDE_HRFS_ACCESS_TIMESTAMP component. The component is included in the BUNDLE_RTP_POSIX_PSE52 component bundle.

For access time stamps to be saved to disk, the volume must be formatted with HRFS on-disk format 1.2 or greater. Version 1.2 is the default version for VxWorks.
6.3. See API references for hrfsAdvFormat() and hrfsAdvFormatFd() for more information.

When the access timestamp component is included, and the appropriate disk format version is used, reading from a file or directory causes its access time stamp to be updated. This can cause significant performance loss, as a write to disk occurs even on a read operation and a transaction point is set. Only use access time stamps if the application requires it for POSIX compliance.

7.4.6 Maximum Number of Files and Directories

HRFS files and directories are stored on disk in data structures called inodes. During formatting the maximum number of inodes is specified as a parameter to hrfsFormat(). The total number of files and directories can never exceed the number inodes. Attempting to create a file or directory when all inodes are in use generates an error. Deleting a file or directory frees the corresponding inode.

7.4.7 Working with Directories

This section discusses creating and removing directories, and reading directory entries.

Creating Subdirectories

You can create as many subdirectories as there are inodes. Subdirectories can be created in the following ways:

- With open(). To create a directory, the O_CREAT option must be set in the flags parameter and the S_IFDIR or FSTAT_DIR option must be set in the mode parameter. The open() calls returns a file descriptor that describes the new directory. The file descriptor can only be used for reading only and should be closed when it no longer needed.

- With mkdir() from usrFsLib.

When creating a directory using either of the above methods, the new directory name must be specified. This name can be either a full pathname or a pathname relative to the current working directory.
Removing Subdirectories

A directory that is to be deleted must be empty (except for the "." and ".." entries). The root directory can never be deleted. Subdirectories can be removed in the following ways:

- Using `ioctl()` with the FIORMDIR function and specifying the name of the directory. The file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
- Using the `remove()` function, specifying the name of the directory.
- Use `rmdir()` from `usrFsLib`.

Reading Directory Entries

You can programmatically search directories on HRFS volumes using the `opendir()`, `readdir()`, `rewinddir()`, and `closedir()` routines.

To obtain more detailed information about a specific file, use the `fstat()` or `stat()` routine. Along with standard file information, the structure used by these routines also provides the file-attribute byte from a directory entry.

For more information, see the API reference for `dirLib`.

7.4.8 Working with Files

This section discusses file I/O and file attributes.

File I/O Routines

Files on an HRFS file system device are created, deleted, written, and read using the standard VxWorks I/O routines: `creat()`, `remove()`, `write()`, and `read()`. For more information, see 6.3 Basic I/O, p.263, and the `ioLib` API references.

Note that and `remove()` is synonymous with `unlink()` for HRFS.

File Linking and Unlinking

When a link is created an inode is not used. Another directory entry is created at the location specified by the parameter to `link()`. In addition, a reference count to
the linked file is stored in the file’s corresponding inode. When unlinking a file, this reference count is decremented. If the reference count is zero when `unlink()` is called, the file is deleted except if there are open file descriptors open on the file. In this case the directory entry is removed but the file still exists on the disk. This prevents tasks and processes (RTPs) from opening the file. When the final open file descriptor is closed the file is fully deleted freeing its inode.

Note that you cannot create a link to a subdirectory only to a regular file.

**File Permissions**

HRFS files have POSIX-style permission bits (unlike dosFs files, which have attributes). The bits can be changed using the `chmod()` and `fchmod()` routines. See the API references for more information.

**7.4.9 Crash Recovery and Volume Consistency**

For detailed information about crash recovery and volume consistence, see *VxWorks Kernel Programmer’s Guide: Local File Systems*.

**7.4.10 I/O Control Functions Supported by HRFS**

The HRFS file system supports the `ioctl()` functions. These functions are defined in the header file `ioLib.h` along with their associated constants; and they are listed in Table 7-1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Decimal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIODISKCHANGE</td>
<td>13</td>
<td>Announces a media change.</td>
</tr>
<tr>
<td>FIODISKFORMAT</td>
<td>5</td>
<td>Formats the disk (device driver function).</td>
</tr>
<tr>
<td>FIODISKINIT</td>
<td>6</td>
<td>Initializes a file system on a disk volume.</td>
</tr>
<tr>
<td>FIOFLUSH</td>
<td>2</td>
<td>Flushes the file output buffer.</td>
</tr>
<tr>
<td>FIOFSTATGET</td>
<td>38</td>
<td>Gets file status information (directory entry data).</td>
</tr>
<tr>
<td>Function</td>
<td>Decimal Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>18</td>
<td>Gets the filename of the <em>fd</em>.</td>
</tr>
<tr>
<td>FIONMOVE</td>
<td>47</td>
<td>Moves a file (does not rename the file).</td>
</tr>
<tr>
<td>FIONFREE</td>
<td>30</td>
<td>Gets the number of free bytes on the volume.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>1</td>
<td>Gets the number of unread bytes in a file.</td>
</tr>
<tr>
<td>FIOREADDIR</td>
<td>37</td>
<td>Reads the next directory entry.</td>
</tr>
<tr>
<td>FIORENAME</td>
<td>10</td>
<td>Renames a file or directory.</td>
</tr>
<tr>
<td>FIORMDIR</td>
<td>32</td>
<td>Removes a directory.</td>
</tr>
<tr>
<td>FIOSEEK</td>
<td>7</td>
<td>Sets the current byte offset in a file.</td>
</tr>
<tr>
<td>FIOSYNC</td>
<td>21</td>
<td>Same as FIOFLUSH, but also re-reads buffered file data.</td>
</tr>
<tr>
<td>FIOTRUNC</td>
<td>42</td>
<td>Truncates a file to a specified length.</td>
</tr>
<tr>
<td>FIOUNMOUNT</td>
<td>39</td>
<td>Un-mounts a disk volume.</td>
</tr>
<tr>
<td>FioWHERE</td>
<td>8</td>
<td>Returns the current byte position in a file.</td>
</tr>
<tr>
<td>FIONCONTIG64</td>
<td>50</td>
<td>Gets the maximum contiguous disk space into a 64-bit integer.</td>
</tr>
<tr>
<td>FIONFREE64</td>
<td>51</td>
<td>Gets the number of free bytes into a 64-bit integer.</td>
</tr>
<tr>
<td>FIONREAD64</td>
<td>52</td>
<td>Gets the number of unread bytes in a file into a 64-bit integer.</td>
</tr>
<tr>
<td>FIOSEEK64</td>
<td>53</td>
<td>Sets the current byte offset in a file from a 64-bit integer.</td>
</tr>
<tr>
<td>FIOWHERE64</td>
<td>54</td>
<td>Gets the current byte position in a file into a 64-bit integer.</td>
</tr>
<tr>
<td>FIOTRUNC64</td>
<td>55</td>
<td>Set the file’s size from a 64-bit integer.</td>
</tr>
</tbody>
</table>

For more information, see the API reference for `ioctl()` in `ioLib`. 
7.5 MS-DOS-Compatible File System: dosFs

The dosFs file system is an MS-DOS-compatible file system that offers considerable flexibility appropriate to the multiple demands of real-time applications. The primary features are:

- Hierarchical files and directories, allowing efficient organization and an arbitrary number of files to be created on a volume.
- A choice of contiguous or non-contiguous files on a per-file basis.
- Compatible with widely available storage and retrieval media (diskettes, hard drives, and so on).
- The ability to boot VxWorks from a dosFs file system.
- Support for VFAT (Microsoft VFAT long file names)
- Support for FAT12, FAT16, and FAT32 file allocation table types.

For information about dosFs libraries, see the VxWorks API references for dosFsLib and dosFsFmtLib.

For information about the MS-DOS file system, please see the Microsoft documentation.

dosFs and Flash Memory

For information about using dosFs with flash memory, see the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS.

dosFs and the Transaction-Based Reliable File System Facility

The dosFs file system can be used with the transaction-based reliable file system (TRFS) facility; see 6.7.7 Transaction-Based Reliable File System Facility: TRFS, p.287.

7.5.1 Configuring VxWorks for dosFs

To include dosFs support in VxWorks, configure the kernel with the appropriate required and optional components.
Required Components

The following components are required:

- INCLUDE_DOSFS_MAIN: dosFsLib
- INCLUDE_DOSFS.Fat: dosFs FAT12/16/32 FAT handler
- INCLUDE_XBD: XBD component

And, either one or both of the following components are required:

- INCLUDE_DOSFS_DIR_VFAT: Microsoft VFAT direct handler
- INCLUDE_DOSFS_DIR_FIXED: Strict 8.3 & VxLongNames directory handler

In addition, you must include the appropriate component for your block device; for example, INCLUDE_ATA.

If you are using a device driver that is not designed for use with the XBD facility, you must use the INCLUDE_XBD_BLK_DEV wrapper component in addition to INCLUDE_XBD. See the VxWorks Kernel Programmer’s Guide: I/O System for more information.

Note that you can use INCLUDE_DOSFS to automatically include the following components:

- INCLUDE_DOSFS_MAIN
- INCLUDE_DOSFS_DIR_VFAT
- INCLUDE_DOSFS_DIR_FIXED
- INCLUDE_DOSFS_FAT
- INCLUDE_DOSFS_CHKDSK
- INCLUDE_DISK_UTIL

Optional dosFs Components

The optional dosFs components are:

- INCLUDE_DOSFS_CACHE: disk cache facility
- INCLUDE_DOSFS_FMT: dosFs file system formatting module
- INCLUDE_DOSFS_CHKDSK: file system integrity checking
- INCLUDE_DISK_UTIL: standard file system operations, such as ls, cd, mkdir, xcopy, and so on
- INCLUDE_TAR: the tar utility
Optional XBD Components

Optional XBD components are:

- INCLUDE_XBD_PART_LIB  disk partitioning facilities
- INCLUDE_XBD_TRANS  TRFS support facility
- INCLUDE_XBD_RAMDRV  RAM disk facility

For information about the XBD facility, see the VxWorks Kernel Programmer’s Guide: I/O System.

7.5.2 Creating a dosFs File System

For information about creating a dosFs file system, see VxWorks Kernel Programmer’s Guide: Local File Systems.

7.5.3 Working with Volumes and Disks

This section discusses accessing volume configuration information and synchronizing volumes. For information about ioctl() support functions, see 7.5.8 I/O Control Functions Supported by dosFsLib, p.314.

Accessing Volume Configuration Information

The dosFsShow() routine can be used to display volume configuration information from the shell. The dosFsVolDescGet() routine can be used programmatically obtain or verify a pointer to the DOS_VOLUME_DESC structure. For more information, see the API references for these routines.

Synchronizing Volumes

When a disk is synchronized, all modified buffered data is physically written to the disk, so that the disk is up to date. This includes data written to files, updated directory information, and the FAT. To avoid loss of data, a disk should be synchronized before it is removed. For more information, see the API references for close() and dosFsVolUnmount().
7.5.4 Working with Directories

This section discusses creating and removing directories, and reading directory entries.

Creating Subdirectories

For FAT32, subdirectories can be created in any directory at any time. For FAT12 and FAT16, subdirectories can be created in any directory at any time, except in the root directory once it reaches its maximum entry count. Subdirectories can be created in the following ways:

- Using ioctl() with the FIOMKDIR function. The name of the directory to be created is passed as a parameter to ioctl().
- Using open(). To create a directory, the O_CREAT option must be set in the flags parameter and the FSTAT_DIR option must be set in the mode parameter. The open() call returns a file descriptor that describes the new directory. Use this file descriptor for reading only, and close it when it is no longer needed.
- Use mkdir() from usrFsLib.

When creating a directory using any of the above methods, the new directory name must be specified. This name can be either a full pathname or a pathname relative to the current working directory.

Removing Subdirectories

A directory that is to be deleted must be empty (except for the “.” and “..” entries). The root directory can never be deleted. Subdirectories can be removed in the following ways:

- Using ioctl() with the FIORMDIR function, specifying the name of the directory. The file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
- Using the remove() function, specifying the name of the directory.
- Use rmdir() from usrFsLib.
Reading Directory Entries

You can programmatically search directories on dosFs volumes using the `opendir()`, `readdir()`, `rewinddir()`, and `closedir()` routines.

To obtain more detailed information about a specific file, use the `fstat()` or `stat()` routine. Along with standard file information, the structure used by these routines also returns the file-attribute byte from a directory entry. For more information, see the API reference for `dirLib`.

7.5.5 Working with Files

This section discusses file I/O and file attributes.

File I/O Routines

Files on a dosFs file system device are created, deleted, written, and read using the standard VxWorks I/O routines: `creat()`, `remove()`, `write()`, and `read()`. For more information, see 6.3 Basic I/O, p.263, and the `ioLib` API references.

File Attributes

The file-attribute byte in a dosFs directory entry consists of a set of flag bits, each indicating a particular file characteristic. The characteristics described by the file-attribute byte are shown in Table 7-2.

<table>
<thead>
<tr>
<th>VxWorks Flag Name</th>
<th>Hex Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOS_ATTR_RDONLY</td>
<td>0x01</td>
<td>read-only file</td>
</tr>
<tr>
<td>DOS_ATTR_HIDDEN</td>
<td>0x02</td>
<td>hidden file</td>
</tr>
<tr>
<td>DOS_ATTR_SYSTEM</td>
<td>0x04</td>
<td>system file</td>
</tr>
<tr>
<td>DOS_ATTR_VOL_LABEL</td>
<td>0x08</td>
<td>volume label</td>
</tr>
<tr>
<td>DOS_ATTR_DIRECTORY</td>
<td>0x10</td>
<td>subdirectory</td>
</tr>
<tr>
<td>DOS_ATTR_ARCHIVE</td>
<td>0x20</td>
<td>file is subject to archiving</td>
</tr>
</tbody>
</table>
7.5 MS-DOS-Compatible File System: dosFs

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 FIO_LABELGET.
The volume label can be set (or reset) to any string of 11 or fewer characters,
using the ioctl() function FIO_LABELSET. 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 FIO_ATTRIBSET. This function is called
after the opening of the specific file with the attributes to be changed. The
attribute-byte value specified in the FIO_ATTRIBSET call is copied directly; to
preserve existing flag settings, determine the current attributes using stat() or
fstat(), then change them using bitwise AND and OR operators.
Example 7-1  Setting DosFs File Attributes

This example makes a dosFs file read-only, and leaves other attributes intact.

```c
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`.

7.5.6 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.
7 Local File Systems
7.5 MS-DOS-Compatible File System: dosFs

- **cluster group allocation (nearly contiguous)**

  *Cluster group allocation* uses adjacent (contiguous) groups of clusters, called *extents*. Cluster group allocation is nearly contiguous allocation and is the default method used when files are written in units larger than the size of a disk’s cluster.

- **absolutely contiguous allocation**

  *Absolutely contiguous allocation* uses only absolutely contiguous clusters. Because this type of allocation is dependent upon the existence of such space, it is specified under only two conditions: immediately after a new file is created and when reading from a file assumed to have been allocated to a contiguous space. Using this method risks disk fragmentation.

For any allocation method, you can deallocate unused reserved bytes by using the POSIX-compliant routine `ftruncate()` or the `ioctl()` function `FIOTRUNC`.

### Choosing an Allocation Method

Under most circumstances, cluster group allocation is preferred to absolutely contiguous file access. Because it is nearly contiguous file access, it achieves a nearly optimal access speed. Cluster group allocation also significantly minimizes the risk of fragmentation posed by absolutely contiguous allocation.

Absolutely contiguous allocation attains raw disk throughput levels, however this speed is only slightly faster than nearly contiguous file access. Moreover, fragmentation is likely to occur over time. This is because after a disk has been in use for some period of time, it becomes impossible to allocate contiguous space. Thus, there is no guarantee that new data, appended to a file created or opened with absolutely continuous allocation, will be contiguous to the initially written data segment.

It is recommended that for a performance-sensitive operation, the application regulate disk space utilization, limiting it to 90% of the total disk space. Fragmentation is unavoidable when filling in the last free space on a disk, which has a serious impact on performance.

### Using Cluster Group Allocation

The dosFs file system defines the size of a cluster group based on the media’s physical characteristics. That size is fixed for each particular media. Since seek operations are an overhead that reduces performance, it is desirable to arrange
files so that sequential portions of a file are located in physically contiguous disk clusters. Cluster group allocation occurs when the cluster group size is considered sufficiently large so that the seek time is negligible compared to the read/write time. This technique is sometimes referred to as nearly contiguous file access because seek time between consecutive cluster groups is significantly reduced.

Because all large files on a volume are expected to have been written as a group of extents, removing them frees a number of extents to be used for new files subsequently created. Therefore, as long as free space is available for subsequent file storage, there are always extents available for use. Thus, cluster group allocation effectively prevents fragmentation (where a file is allocated in small units spread across distant locations on the disk). Access to fragmented files can be extremely slow, depending upon the degree of fragmentation.

Using Absolutely Contiguous Allocation

A contiguous file is made up of a series of consecutive disk sectors. Absolutely contiguous allocation is intended to allocate contiguous space to a specified file (or directory) and, by so doing, optimize access to that file. You can specify absolutely contiguous allocation either when creating a file, or when opening a file previously created in this manner.

For more information on the ioctl() functions, see 7.5.8 I/O Control Functions Supported by dosFsLib, p.314.

Allocating Contiguous Space for a File

To allocate a contiguous area to a newly created file, follow these steps:

1. First, create the file in the normal fashion using open() or creat().
2. Then, call ioctl(). Use the file descriptor returned from open() or creat() as the file descriptor argument. Specify FIOCONTIG as the function code argument and the size of the requested contiguous area, in bytes, as the third argument.

The FAT is then searched for a suitable section of the disk. If found, this space is assigned to the new file. The file can then be closed, or it can be used for further I/O operations. The file descriptor used for calling ioctl() should be the only descriptor open to the file. Always perform the ioctl() FIOCONTIG operation before writing any data to the file.

To request the largest available contiguous space, use CONTIG_MAX for the size of the contiguous area. For example:

```
status = ioctl (fd, FIOCONTIG, CONTIG_MAX);
```
Allocating Space for Subdirectories

Subdirectories can also be allocated a contiguous disk area in the same manner:

- If the directory is created using the `ioctl()` function `FIOMKDIR`, it must be subsequently opened to obtain a file descriptor to it.
- If the directory is created using options to `open()`, the returned file descriptor from that call can be used.

A directory must be empty (except for the “.” and “..” entries) when it has contiguous space allocated to it.

Opening and Using a Contiguous File

Fragmented files require following cluster chains in the FAT. However, if a file is recognized as contiguous, the system can use an enhanced method that improves performance. This applies to all contiguous files, whether or not they were explicitly created using `FIOCONTIG`. Whenever a file is opened, it is checked for contiguity. If it is found to be contiguous, the file system registers the necessary information about that file to avoid the need for subsequent access to the FAT table. This enhances performance when working with the file by eliminating seek operations.

When you are opening a contiguous file, you can explicitly indicate that the file is contiguous by specifying the `DOS_O_CONTIG_CHK` flag with `open()`. This prompts the file system to retrieve the section of contiguous space, allocated for this file, from the FAT table.

To find the maximum contiguous area on a device, you can use the `ioctl()` function `FIONCONTIG`. This information can also be displayed by `dosFsConfigShow()`. 
Example 7-2  **Finding the Maximum Contiguous Area on a DosFs Device**

In this example, the size (in bytes) of the largest contiguous area is copied to the integer pointed to by the third parameter to `ioctl()` (`count`).

```c
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);
}
```

### 7.5.7 Crash Recovery and Volume Consistency

For information about crash recovery and volume consistence, see *VxWorks Kernel Programmer’s Guide: Local File Systems*.

### 7.5.8 I/O Control Functions Supported by dosFsLib

The dosFs file system supports the `ioctl()` functions. These functions are defined in the header file `ioLib.h` along with their associated constants, and they are described in Table 7-3.
## Local File Systems

### 7.5 MS-DOS-Compatible File System: dosFs

<table>
<thead>
<tr>
<th>Function</th>
<th>Decimal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIOATTRIBSET</td>
<td>35</td>
<td>Sets the file-attribute byte in the dosFs directory entry.</td>
</tr>
<tr>
<td>FIOCONTIG</td>
<td>36</td>
<td>Allocates contiguous disk space for a file or directory.</td>
</tr>
<tr>
<td>FIODISKCHANGE</td>
<td>13</td>
<td>Announces a media change.</td>
</tr>
<tr>
<td>FIODISKFORMAT</td>
<td>5</td>
<td>Formats the disk (device driver function).</td>
</tr>
<tr>
<td>FIODISKINIT</td>
<td>6</td>
<td>Initializes a dosFs file system on a disk volume.</td>
</tr>
<tr>
<td>FIOFLUSH</td>
<td>2</td>
<td>Flushes the file output buffer.</td>
</tr>
<tr>
<td>FIOFSTATGET</td>
<td>38</td>
<td>Gets file status information (directory entry data).</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>18</td>
<td>Gets the filename of the $fd$.</td>
</tr>
<tr>
<td>FIOLABELGET</td>
<td>33</td>
<td>Gets the volume label.</td>
</tr>
<tr>
<td>FIOLABELSET</td>
<td>34</td>
<td>Sets the volume label.</td>
</tr>
<tr>
<td>FIoMKDIR</td>
<td>31</td>
<td>Creates a new directory.</td>
</tr>
<tr>
<td>FIOMOVE</td>
<td>47</td>
<td>Moves a file (does not rename the file).</td>
</tr>
<tr>
<td>FIONCONTIG</td>
<td>41</td>
<td>Gets the size of the maximum contiguous area on a device.</td>
</tr>
<tr>
<td>FIONFREE</td>
<td>30</td>
<td>Gets the number of free bytes on the volume.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>1</td>
<td>Gets the number of unread bytes in a file.</td>
</tr>
<tr>
<td>FIOREADDIR</td>
<td>37</td>
<td>Reads the next directory entry.</td>
</tr>
<tr>
<td>FIORENAME</td>
<td>10</td>
<td>Renames a file or directory.</td>
</tr>
<tr>
<td>FIORMDIR</td>
<td>32</td>
<td>Removes a directory.</td>
</tr>
<tr>
<td>FIOSEEK</td>
<td>7</td>
<td>Sets the current byte offset in a file.</td>
</tr>
<tr>
<td>FIOSYNC</td>
<td>21</td>
<td>Same as FIOFLUSH, but also re-reads buffered file data.</td>
</tr>
<tr>
<td>FIOTRUNC</td>
<td>42</td>
<td>Truncates a file to a specified length.</td>
</tr>
</tbody>
</table>
For more information, see the API references for `dosFsLib` and `ioctl()` in `ioLib`.

### 7.5.9 Booting from a Local dosFs File System Using SCSI

For information about booting from a local dosFs file system using SCSI, see *VxWorks Kernel Programmer’s Guide: Local File Systems*.

### 7.6 Raw File System: rawFs

VxWorks provides a raw file system (rawFs) for use in systems that require only the most basic disk I/O functions. The rawFs file system, implemented with `rawFsLib`, treats the entire disk volume much like a single large file.
Although the dosFs file system provides this ability to varying degrees, the rawFs file system offers advantages in size and performance if more complex functions are not required.

The rawFs file system imposes no organization of the data on the disk. It maintains no directory information; and there is therefore no division of the disk area into specific files. All open() operations on rawFs devices specify only the device name; no additional filenames are possible.

The entire disk area is treated as a single file and is available to any file descriptor that is open for the device. All read and write operations to the disk use a byte-offset relative to the start of the first block on the disk.

A rawFs file system is created by default if inserted media does not contain a recognizable file system.

### 7.6.1 Configuring VxWorks for rawFs

To use the rawFs file system, configure VxWorks with the INCLUDE_RAWFS and INCLUDE_XBD components.

If you are using a device driver that is not designed for use with the XBD facility, you must use the INCLUDE_XBD_BLK_DEV wrapper component in addition to INCLUDE_XBD. See the VxWorks Kernel Programmer’s Guide: I/O System for more information.

Set the NUM_RAWFS_FILES parameter of the INCLUDE_RAWFS component to the desired maximum open file descriptor count. For information about using multiple file descriptors with what is essentially a single large file, see 7.6.4 rawFs File I/O, p.318.

### 7.6.2 Creating a rawFs File System

For information about creating a rawFs file system, see VxWorks Kernel Programmer’s Guide: Local File Systems.

### 7.6.3 Mounting rawFs Volumes

A disk volume is mounted automatically, generally during the first open() or creat() operation. (Certain ioctl() functions also cause the disk to be mounted.)
The volume is again mounted automatically on the first disk access following a ready-change operation.

⚠️ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (/) alone as a name or unexpected results may occur.

### 7.6.4 rawFs File I/O

To begin I/O operations upon a rawFs device, first open the device using the standard `open()` routine (or the `creat()` routine). Data on the rawFs device is written and read using the standard I/O routines `write()` and `read()`. For more information, see [6.3 Basic I/O](#), p.263.

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.

### 7.6.5 I/O Control Functions Supported by rawFsLib

The rawFs file system supports the `ioctl()` functions shown in Table 7-4. The functions listed are defined in the header file `ioLib.h`. For more information, see the API references for `rawFsLib` and for `ioctl()` in `ioLib`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Decimal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIODISKCHANGE</td>
<td>13</td>
<td>Announces a media change.</td>
</tr>
<tr>
<td>FIODISKFORMAT</td>
<td>5</td>
<td>Formats the disk (device driver function).</td>
</tr>
<tr>
<td>FIOFLUSH</td>
<td>2</td>
<td>Same as <code>FIOSYNC</code>.</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>18</td>
<td>Gets the device name of the <code>fd</code>.</td>
</tr>
</tbody>
</table>
7 Local File Systems

7.7 CD-ROM File System: cdromFs

The VxWorks CD-ROM file system, cdromFs, allows applications to read data from CDs formatted according to the ISO 9660 standard file system with or without the Joliet extensions. This section describes how cdromFs is organized, configured, and used.

The cdromFs library, cdromFsLib, lets applications read any CD-ROMs, CD-Rs, or CD-RWs (collectively called CDs) that are formatted in accordance with the ISO 9660 file system standard, with or without the Joliet extensions. ISO 9660 interchange level 3, implementation level 2, is supported. Note that multi-extent files, interleaved files, and files with extended attribute records are supported.

The following CD features and ISO 9660 features are not supported:

- Multi-volume sets
- Record format files
- CDs with a sector size that is not a power of two
- Multi-session CD-R or CD-RW

---

1. Therefore, mode 2/form 2 sectors are not supported, as they have 2324 bytes of user data per sector. Both mode 1/form 1 and mode 2/form 1 sectors are supported, as they have 2048 bytes of user data per sector.

2. The first session (that is, the earliest session) is always read. The most commonly desired behavior is to read the last session (that is, the latest session).
After initializing cdromFs and mounting it on a CD-ROM block device, you can access data on that device using the standard POSIX I/O calls: `open()`, `close()`, `read()`, `ioctl()`, `readdir()`, and `stat()`. The `write()` call always returns an error.

The cdromFs utility supports multiple drives, multiple open files, and concurrent file access. When you specify a pathname, cdromFS accepts both forward slashes (/) and back slashes (\) as path delimiters. However, the backslash is not recommended because it might not be supported in future releases.

The initialization sequence for the cdromFs file system is similar to installing a dosFs file system on a SCSI or ATA device.

After you have created the CD file system device (7.7.2 Creating and Using cdromFs, p.321), use `ioctl()` to set file system options. The files system options are described below:

**CDROMFS_DIR_MODE_SET/GET**
These options set and get the directory mode. The directory mode controls whether a file is opened with the Joliet extensions, or without them. The directory mode can be set to any of the following:

- **MODE_ISO9660**
  Do not use the Joliet extensions.

- **MODE_JOLIET**
  Use the Joliet extensions.

- **MODE_AUTO**
  Try opening the directory first without Joliet, and then with Joliet.

⚠️ **CAUTION:** Changing the directory mode un-mounts the file system. Therefore, any open file descriptors are marked as obsolete.

**CDROMFS_STRIP_SEMICOLON**
This option sets the `readdir()` strip semicolon setting to FALSE if `arg` is 0, and to TRUE otherwise. If TRUE, `readdir()` removes the semicolon and following version number from the directory entries retrieved.

**CDROMFS_GET_VOL_DESC**
This option returns, in `arg`, the primary or supplementary volume descriptor by which the volume is mounted. `arg` must be of type `T_ISO_PVD_SVD_ID`, as defined in `cdromFsLib.h`. The result is the volume descriptor, adjusted for the endianness of the processor (not the raw volume descriptor from the CD). This result can be used directly by the processor. The result also includes some information not in the volume descriptor, such as which volume descriptor is in use.
For information on using `cdromFs()`, see the API reference for `cdromFsLib`.

### 7.7.1 Configuring VxWorks for cdromFs

To configure VxWorks with cdromFs, add the `INCLUDE_CDROMFS` and `INCLUDE_XBD` components to the kernel. Add other required components (such as SCSI or ATA) depending on the type of device).

If you are using a device driver that is not designed for use with the XBD facility, you must use the `INCLUDE_XBD_BLK_DEV` wrapper component in addition to `INCLUDE_XBD`. See the VxWorks Kernel Programmer’s Guide: I/O System for more information.

If you are using an ATAPI device, make appropriate modifications to the `ataDrv`, `ataResources[ ]` structure array (if needed). This must be configured appropriately for your hardware platform.

### 7.7.2 Creating and Using cdromFs

For information about creating and using a CD block device, see *VxWorks Kernel Programmer’s Guide: Local File Systems*. 
7.7.3 I/O Control Functions Supported by cdromFsLib

The cdromFs file system supports the `ioctl()` functions. These functions, and their associated constants, are defined in the header files `ioLib.h` and `cdromFsLib.h`. Table 7-5 describes the `ioctl()` functions that `cdromFsLib` supports. For more information, see the API references for `cdromFsLib` and for `ioctl()` in `ioLib`.

### Table 7-5 `ioctl()` Functions Supported by cdromFsLib

<table>
<thead>
<tr>
<th>Function Constant</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDROMFS_DIR_MODE_GET</td>
<td>7602176</td>
<td>Gets the volume descriptor(s) used to open files.</td>
</tr>
<tr>
<td>CDROMFS_DIR_MODE_SET</td>
<td>7602177</td>
<td>Sets the volume descriptor(s) used to open files.</td>
</tr>
<tr>
<td>CDROMFS_GET_VOL_DESC</td>
<td>7602179</td>
<td>Gets the volume descriptor that is currently in use.</td>
</tr>
<tr>
<td>CDROMFS_STRIP_SEMICOLON</td>
<td>7602178</td>
<td>Sets the <code>readdir()</code> strip version number setting.</td>
</tr>
<tr>
<td>FIOFSTATGET</td>
<td>38</td>
<td>Gets file status information (directory entry data).</td>
</tr>
<tr>
<td>FIOGETNAME</td>
<td>18</td>
<td>Gets the filename of the file descriptor.</td>
</tr>
<tr>
<td>FIOLABELGET</td>
<td>33</td>
<td>Gets the volume label.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>1</td>
<td>Gets the number of unread bytes in a file.</td>
</tr>
<tr>
<td>FIONREAD64</td>
<td>52</td>
<td>Gets the number of unread bytes in a file (64-bit version).</td>
</tr>
<tr>
<td>FIOREADDIR</td>
<td>37</td>
<td>Reads the next directory entry.</td>
</tr>
<tr>
<td>FIOSEEK</td>
<td>7</td>
<td>Sets the current byte offset in a file.</td>
</tr>
<tr>
<td>FIOSEEK64</td>
<td>53</td>
<td>Sets the current byte offset in a file (64-bit version).</td>
</tr>
<tr>
<td>FIOUNMOUNT</td>
<td>39</td>
<td>Un-mounts a disk volume.</td>
</tr>
<tr>
<td>FIOWHERE</td>
<td>8</td>
<td>Returns the current byte position in a file.</td>
</tr>
<tr>
<td>FIOWHERE64</td>
<td>54</td>
<td>Returns the current byte position in a file (64-bit version).</td>
</tr>
</tbody>
</table>
7 Local File Systems

7.8 Read-Only Memory File System: ROMFS

7.7.4 Version Numbers

cdromFsLib has a 4-byte version number. The version number is composed of four parts, from most significant byte to least significant byte:

- major number
- minor number
- patch level
- build

The version number is returned by cdromFsVersionNumGet() and displayed by cdromFsVersionNumDisplay().

7.8 Read-Only Memory File System: ROMFS

ROMFS is a simple, read-only file system that represents and stores files and directories in a linear way (similar to the tar utility). It is installed in RAM with the VxWorks system image at boot time. The name ROMFS stands for Read-Only Memory File System; it does not imply any particular relationship to ROM media.

ROMFS provides the ability to bundle VxWorks applications—or any other files for that matter—with the operating system. No local disk or network connection to a remote disk is required for executables or other files. When VxWorks is configured with the ROMFS component, files of any type can be included in the operating system image simply by adding them to a ROMFS directory on the host system, and then rebuilding VxWorks. The build produces a single system image that includes both the VxWorks and the files in the ROMFS directory.

When VxWorks is booted with this image, the ROMFS file system and the files it holds are loaded with the kernel itself. ROMFS allows you to deploy files and operating system as a unit. In addition, process-based applications can be coupled with an automated startup facility so that they run automatically at boot time. ROMFS thereby provides the ability to create fully autonomous, multi-process systems.

ROMFS can also be used to store applications that are run interactively for diagnostic purposes, or for applications that are started by other applications under specific conditions (errors, and so on).
7.8.1 Configuring VxWorks with ROMFS

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

7.8.2 Building a System With ROMFS and Files

Configuring VxWorks with ROMFS and applications involves the following steps:

1. Create a ROMFS directory in the project directory on the host system, using the name /romfs.
2. Copy the application files into the /romfs directory.
3. Rebuild VxWorks.

For example, adding a process-based application called myVxApp.vxe from the command line would look like this:

```
cd c:\myInstallDir\vxworks-6.1\target\proj\wrSbc8260_diab
mkdir romfs
copy c:\allMyVxApps\myVxApp.vxe romfs
make TOOL=diab
```

The contents of the romfs directory are automatically built into a ROMFS file system and combined with the VxWorks image.

The ROMFS directory does not need to be created in the VxWorks project directory. It can also be created in any location on (or accessible from) the host system, and the make utility’s ROMFS_DIR macro used to identify where it is in the build command. For example:

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

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

7.8.3 Accessing Files in ROMFS

At run-time, the ROMFS file system is accessed as /romfs. The content of the ROMFS directory can be browsed using the ls and cd shell commands, and accessed programmatically with standard file system routines, such as open() and read().

For example, if the directory

```
installDir/vxworks-6.x/target/proj/wrSbc8260_diab/romfs
```

has been created on the
host, the file foo copied to it, and the system rebuilt and booted; then using cd and ls from the shell (with the command interpreter) looks like this:

```
[vxWorks *]# cd /romfs
[vxWorks *]# ls
.
..
foo
[vxWorks *]#
```

And foo can also be accessed at run-time as /romfs/foo by any applications running on the target.

### 7.8.4 Using ROMFS to Start Applications Automatically

ROMFS can be used with various startup mechanisms to start process-based applications automatically when VxWorks boots.

See [2.9.4 Using ROMFS to Start Applications Automatically](#), p.75 for more information.

### 7.9 Target Server File System: TSFS

The Target Server File System (TSFS) is designed for development and diagnostic purposes. It is a full-featured VxWorks file system, but the files are actually located on the host system.

The TSFS provides all of the I/O features of the network driver for remote file access (see [6.7.5 Non-NFS Network Devices](#), p.286), without requiring any target resources—except those required for communication between the target system and the target server on the host. The TSFS uses a WDB target agent driver to transfer requests from the VxWorks I/O system to the target server. The target server reads the request and executes it using the host file system. When you open a file with TSFS, the file being opened is actually on the host. Subsequent `read()` and `write()` calls on the file descriptor obtained from the `open()` call read from and write to the opened host file.

The TSFS VIO driver is oriented toward file I/O rather than toward console operations. TSFS provides all the I/O features that `netDrv` provides, without requiring any target resource beyond what is already configured to support
communication between target and target server. It is possible to access host files randomly without copying the entire file to the target, to load an object module from a virtual file source, and to supply the filename to routines such as `ld()` and `copy()`.

Each I/O request, including `open()`, is synchronous; the calling target task is blocked until the operation is complete. This provides flow control not available in the console VIO implementation. In addition, there is no need for WTX protocol requests to be issued to associate the VIO channel with a particular host file; the information is contained in the name of the file.

Consider a `read()` call. The driver transmits the ID of the file (previously established by an `open()` call), the address of the buffer to receive the file data, and the desired length of the read to the target server. The target server responds by issuing the equivalent `read()` call on the host and transfers the data read to the target program. The return value of `read()` and any `errno` that might arise are also relayed to the target, so that the file appears to be local in every way.

For detailed information, see the API reference for `wdbTsfsDrv`.

**Socket Support**

TSFS sockets are operated on in a similar way to other TSFS files, using `open()`, `close()`, `read()`, `write()`, and `ioctl()`. To open a TSFS socket, use one of the following forms of filename:

```
*TCP:hostIP:port*
*TCP:hostname:port*
```

The `flags` and `permissions` arguments are ignored. The following examples show how to use these filenames:

```c
fd = open("/tgtsvr/TCP:phobos:6164",0,0); /* open socket and connect */
   /* to server phobos */

fd = open("/tgtsvr/TCP:150.50.50.50:6164",0,0); /* open socket and */
   /* connect to server */
   /* 150.50.50.50 */
```

The result of this `open()` call is to open a TCP socket on the host and connect it to the target server socket at `hostname` or `hostIP` awaiting connections on `port`. The resultant socket is non-blocking. Use `read()` and `write()` to read and write to the TSFS socket. Because the socket is non-blocking, the `read()` call returns immediately with an error and the appropriate `errno` if there is no data available to read from the socket. The `ioctl()` usage specific to TSFS sockets is discussed in the API reference for `wdbTsfsDrv`. This socket configuration allows VxWorks to
use the socket facility without requiring sockLib and the networking modules on the target.

Error Handling

Errors can arise at various points within TSFS and are reported back to the original caller on the target, along with an appropriate error code. The error code returned is the VxWorks *errno* which most closely matches the error experienced on the host. If a WDB error is encountered, a WDB error message is returned rather than a VxWorks *errno*.

Configuring VxWorks for TSFS Use

To use TSFS, configure VxWorks with the `INCLUDE_WDB_TSFS` component. This creates the `/tgtsvr` file system on the target.

The target server on the host system must also be configured for TSFS. This involves assigning a root directory on your host to TSFS (see the discussion of the target server `-R` option in Security Considerations, p.327). For example, on a PC host you could set the TSFS root to `c:\myTarget\logs`.

Having done so, opening the file `/tgtsvr/logFoo` on the target causes `c:\myTarget\logs\logFoo` to be opened on the host by the target server. A new file descriptor representing that file is returned to the caller on the target.

Security Considerations

While TSFS has much in common with `netDrv`, the security considerations are different (also see 6.7.5 Non-NFS Network Devices, p.286). With TSFS, the host file operations are done on behalf of the user that launched the target server. The user name given to the target as a boot parameter has no effect. In fact, none of the boot parameters have any effect on the access privileges of TSFS.

In this environment, it is less clear to the user what the privilege restrictions to TSFS actually are, since the user ID and host machine that start the target server may vary from invocation to invocation. By default, any host tool that connects to a target server which is supporting TSFS has access to any file with the same authorizations as the user that started that target server. However, the target server can be locked (with the `-L` option) to restrict access to the TSFS.
The options which have been added to the target server startup routine to control target access to host files using TSFS include:

- **R** Set the root of TSFS.
  For example, specifying `-R /tftpboot` prepends this string to all TSFS filenames received by the target server, so that `/tgtsvr/etc/passwd` maps to `/tftpboot/etc/passwd`. If `-R` is not specified, TSFS is not activated and no TSFS requests from the target will succeed. Restarting the target server without specifying `-R` disables TSFS.

- **RW** Make TSFS read-write.
  The target server interprets this option to mean that modifying operations (including file create and delete or write) are authorized. If `-RW` is not specified, the default is read only and no file modifications are allowed.

**NOTE:** For more information about the target server and the TSFS, see the `tgtsvr` command reference. For information about specifying target server options from Workbench, see the *Wind River Workbench User’s Guide: Setting Up Your Hardware* and the *Wind River Workbench User’s Guide: New Target Server Connections*.

**Using the TSFS to Boot a Target**

For information about using the TSFS to boot a targets, see *VxWorks Kernel Programmer’s Guide: Kernel*.
8.1 Introduction 329

VxWorks provides an error detection and reporting facility to help debugging software faults. It does so by recording software exceptions in a specially designated area of memory that is not cleared between warm reboots. The facility also allows for selecting system responses to fatal errors, with alternate strategies for development and deployed systems.

The key features of the error detection and reporting facility are:

- A persistent memory region in RAM used to retain error records across warm reboots.
Mechanisms for recording various types of error records.
- Error records that provide detailed information about run-time errors and the conditions under which they occur.
- The ability to display error records and clear the error log from the shell.
- Alternative error-handing options for the system’s response to fatal errors.
- Macros for implementing error reporting in user code.

For more information about error detection and reporting routines in addition to that provided in this chapter, see the API reference for edrLib. Also see the VxWorks Kernel Programmer’s Guide: Error Detection and Reporting for information about facilities available only in the kernel.

For information about related facilities, see 5.4 Memory Error Detection, p.248.

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

### 8.2 Configuring Error Detection and Reporting Facilities

To use the error detection and reporting facilities:

- VxWorks must be configured with the appropriate components.
- A persistent RAM memory region must be configured, and it must be sufficiently large to hold the error records.
- Optionally, users can change the system’s default response to fatal errors.

#### 8.2.1 Configuring VxWorks

To use the error detection and reporting facility, the kernel must be configured with the following components:

- INCLUDE_EDR_PM
- INCLUDE_EDR_ERRLOG
- INCLUDE_EDR_SHOW
- INCLUDE_EDR_SYSDBG_FLAG
8.2.2 Configuring the Persistent Memory Region

The persistent-memory region is an area of RAM at the top of system memory specifically reserved for an error records. It is protected by the MMU and the VxWorks `vmLib` facilities. The memory is not cleared by warm reboots, provided a VxWorks 6.x boot loader is used.

A cold reboot always clears the persistent memory region. The `pmInvalidate()` routine can also be used to explicitly destroy the region (making it unusable) so that it is recreated during the next warm reboot.

The persistent-memory area is write-protected when the target system includes an MMU and VxWorks has been configured with MMU support.

The size of the persistent memory region is defined by the `PM_RESERVED_MEM` configuration parameter. By default the size is set to six pages of memory.

By default, the error detection and reporting facility uses one-half of whatever persistent memory is available. If no other applications require persistent memory, the component may be configured to use almost all of it. This can be accomplished by defining `EDR_ERRLOG_SIZE` to be the size of `PM_RESERVED_MEM` less the size of one page of memory.

If you increase the size of the persistent memory region beyond the default, you must create a new boot loader with the same `PM_RESERVED_MEM` value. The memory area between `RAM_HIGH_ADRS` and `sysMemTop()` must be big enough to copy the VxWorks boot loader. If it exceeds the `sysMemTop()` limit, the boot loader may corrupt the area of persistent memory reserved for core dump storage when it loads VxWorks. The boot loader, must therefore be rebuilt with a lower `RAM_HIGH_ADRS` value.

**WARNING:** If the boot loader is not properly configured (as described above), this could lead into corruption of the persistent memory region when the system boots.

The `EDR_RECORD_SIZE` parameter can be used to change the default size of error records. Note that for performance reasons, all records are necessarily the same size.

The `pmShow()` shell command (for the C interpreter) can be used to display the amount of allocated and free persistent memory.

For more information about persistent memory, see the *VxWorks Kernel Programmer’s Guide: Memory Management*, and the `pmLib` API reference.
332

8.2.3 Configuring Responses to Fatal Errors

The error detection and reporting facilities provide for two sets of responses to fatal errors. See 8.5 Fatal Error Handling Options, p.335 for information about these responses, and various ways to select one for a run-time system.

8.3 Error Records

Error records are generated automatically when the system experiences specific kinds of faults. The records are stored in the persistent memory region of RAM in a circular buffer. Newer records overwrite older records when the persistent memory buffer is full.

The records are classified according to two basic criteria:

- event type
- severity level

The event type identifies the context in which the error occurred (during system initialization, or in a process, and so on).

The severity level indicates the seriousness of the error. In the case of fatal errors, the severity level is also associated with alternative system’s responses to the error (see 8.5 Fatal Error Handling Options, p.335).

The event types are defined in Table 8-1, and the severity levels in Table 8-2.

Table 8-1 Event Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>System initialization events.</td>
</tr>
<tr>
<td>BOOT</td>
<td>System boot events.</td>
</tr>
</tbody>
</table>
The information collected depends on the type of events that occur. In general, a complete fault record is recorded. For some events, however, portions of the record are excluded for clarity. For example, the record for boot and reboot events exclude the register portion of the record.

Error records hold detailed information about the system at the time of the event. Each record includes the following generic information:

- date and time the record was generated
- type and severity
- operating system version
- task ID
- process ID, if the failing task in a process
- task name
- process name, if the failing task is in a process
- source file and line number where the record was created
- a free form text message

It also optionally includes the following architecture-specific information:

---

**Table 8-1**  
**Event Types** (cont’d)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REBOOT</td>
<td>System reboot (warm boot) events.</td>
</tr>
<tr>
<td>KERNEL</td>
<td>VxWorks kernel events.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Interrupt handler events.</td>
</tr>
<tr>
<td>RTP</td>
<td>Process environment events.</td>
</tr>
<tr>
<td>USER</td>
<td>Custom events (user defined).</td>
</tr>
</tbody>
</table>

---

**Table 8-2**  
**Severity Levels**

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FATAL</td>
<td>Fatal event.</td>
</tr>
<tr>
<td>NONFATAL</td>
<td>Non-fatal event.</td>
</tr>
<tr>
<td>WARNING</td>
<td>Warning event.</td>
</tr>
<tr>
<td>INFO</td>
<td>Informational event.</td>
</tr>
</tbody>
</table>
### 8.4 Displaying and Clearing Error Records

The edrShow library provides a set of commands for the shell’s C interpreter that are used for displaying the error records created since the persistent memory region was last cleared. See Table 8-3.

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>edrShow()</td>
<td>Show all records.</td>
</tr>
<tr>
<td>edrFatalShow()</td>
<td>Show only FATAL severity level records.</td>
</tr>
<tr>
<td>edrInfoShow()</td>
<td>Show only INFO severity level records.</td>
</tr>
<tr>
<td>edrKernelShow()</td>
<td>Show only KERNEL event type records.</td>
</tr>
<tr>
<td>edrRtpShow()</td>
<td>Show only RTP (process) event type records.</td>
</tr>
<tr>
<td>edrUserShow()</td>
<td>Show only USER event type records.</td>
</tr>
<tr>
<td>edrIntShow()</td>
<td>Show only INTERRUPT event type records.</td>
</tr>
<tr>
<td>edrInitShow()</td>
<td>Show only INIT event type records.</td>
</tr>
<tr>
<td>edrBootShow()</td>
<td>Show only BOOT event type records.</td>
</tr>
<tr>
<td>edrRebootShow()</td>
<td>Show only REBOOT event type records.</td>
</tr>
</tbody>
</table>

The shell’s command interpreter provides comparable commands. See the API references for the shell, or use the help edr command.

In addition to displaying error records, each of the show commands also displays the following general information about the error log:

- memory map
- exception information
- processor registers
- disassembly listing (surrounding the faulting address)
- stack trace
8 Error Detection and Reporting

8.5 Fatal Error Handling Options

- total size of the log
- size of each record
- maximum number of records in the log
- the CPU type
- a count of records missed due to no free records
- the number of active records in the log
- the number of reboots since the log was created

See the edrShow API reference for more information.

8.5 Fatal Error Handling Options

In addition to generating error records, the error detection and reporting facility provides for two modes of system response to fatal errors for each event type:

- debug mode, for lab systems (development)
- deployed mode, for production systems (field)

The difference between these modes is in their response to fatal errors in processes (RTP events). In debug mode, a fatal error in a process results in the process being stopped. In deployed mode, as fatal error in a process results in the process being terminated.

The operative error handling mode is determined by the system debug flag (see 8.5.2 Setting the System Debug Flag, p.336). The default is deployed mode.

Table 8-4 describes the responses in each mode for each of the event types. It also lists the routines that are called when fatal records are created.

The error handling routines are called response to certain fatal errors. Only fatal errors—and no other event types—have handlers associated with them. These handlers are defined in installDir/vxworks-6.x/target/config/comps/src/edrStub.c. Developers can modify the routines in this file to implement different system responses to fatal errors. The names of the routines, however, cannot be changed.
Note that when the debugger is attached to the target, it gains control of the system before the error-handling option is invoked, thus allowing the system to be debugged even if the error-handling option calls for a reboot.

### 8.5.1 Configuring VxWorks with Error Handling Options

In order to provide the option of debug mode error handling for fatal errors, VxWorks must be configured with the `INCLUDE_EDR_SYSDBG_FLAG` component, which it is by default. The component allows a system debug flag to be used to select debug mode, as well as reset to deployed mode (see 8.5.2 Setting the System Debug Flag, p. 336). If `INCLUDE_EDR_SYSDBG_FLAG` is removed from VxWorks, the system defaults to deployed mode (see Table 8-4).

### 8.5.2 Setting the System Debug Flag

How the error detection and reporting facility responds to fatal errors, beyond merely recording the error, depends on the setting of the system debug flag. When the system is configured with the `INCLUDE_EDR_SYSDBG_FLAG` component, the flag can be used to set the handling of fatal errors to either debug mode or deployed mode (the default).

For systems undergoing development, it is obviously desirable to leave the system in a state that can be more easily debugged; while in deployed systems, the aim is to have them recover as best as possible from fatal errors and continue operation.

The debug flag can be set in any of the following ways:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Debug Mode</th>
<th>Deployed Mode (default)</th>
<th>Error Handling Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>Reboot</td>
<td>Reboot</td>
<td><code>edrInit FatalPolicyHandler()</code></td>
</tr>
<tr>
<td>KERNEL</td>
<td>Stop failed task</td>
<td>Stop failed task</td>
<td><code>edrKernelFatalPolicyHandler()</code></td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Reboot</td>
<td>Reboot</td>
<td><code>edrInterruptFatalPolicyHandler()</code></td>
</tr>
<tr>
<td>RTP</td>
<td>Stop process</td>
<td>Delete process</td>
<td><code>edrRtpFatalPolicyHandler()</code></td>
</tr>
</tbody>
</table>

Note that when the debugger is attached to the target, it gains control of the system before the error-handling option is invoked, thus allowing the system to be debugged even if the error-handling option calls for a reboot.
8 Error Detection and Reporting

8.5 Fatal Error Handling Options

- Statically, with boot loader configuration.
- Interactively, at boot time.

When a system boots, the banner displayed on the console displays information about the mode defined by the system debug flag. For example:

```
ED&R Policy Mode: Deployed
```

The modes are identified as **Debug**, **Deployed**, or **Permanently Deployed**. The latter indicates that the `INCLUDE_EDR_SYSDBG_FLAG` component is not included in the system, which means that the mode is deployed and that it cannot be changed to debug.

**Setting the Debug Flag Statically**

The system can be set to either debug mode or deployed mode with the `f` boot loader parameter when a boot loader is configured and built. The value of 0x000 is used to select deployed mode. The value of 0x400 is used to select debug mode. By default, it is set to deployed mode.

To change the default behavior when configuring a new system, change the `f` parameter in the `DEFAULT_BOOT_LINE` definition in `installDir/target/config/bspName/config.h`, which looks like this:

```
#define DEFAULT_BOOT_LINE BOOT_DEV_NAME \
"(0,0)wrSbc8260:vxworks " \ 
"e=192.168.100.51 " \ 
"h=192.168.100.155 " \ 
"g=0.0.0.0 " \ 
"u=anonymous pw=user " \ 
"f=0x00 tn=wrSbc8260"
```

For information about configuring and building boot loaders, see the *VxWorks Kernel Programmer’s Guide: Kernel*.

**Setting the Debug Flag Interactively**

To change the system debug flag interactively, stop the system when it boots. Then use the `c` command at the boot-loader command prompt. Change the value of the `f` parameter: use 0x000 for deployed mode (the default) or to 0x400 for debug mode.
8.6 Other Error Handling Options for Processes

By default, any faults generated by a process are handled by the error detection and reporting facility.

A process can, however, handle its own faults by installing an appropriate signal handler in the process. If a signal handler is installed (for example, SIGSEGV or SIGBUS), the signal handler is run instead of an error record being created and an error handler being called. The signal handler may pass control to the facility if it chooses to by using the edrErrorInject() system call.

For more information about signals, see 3.3.10 Signals, p.149.
8.7 Using Error Reporting APIs in Application Code

The edrLib.h file provides a set of convenient macros that developers can use in their source code to generate error messages (and responses by the system to fatal errors) under conditions of the developers choosing.

The macros have no effect if VxWorks has not been configured with error detection and reporting facilities. Code, therefore, must not be conditionally compiled to make use of these facilities.

The edrLib.h file is in installDir/vxworks-6.x/target/usr/h.

The following macros are provided:

**EDR_USER_INFO_INJECT (trace, msg)**

Creates a record in the error log with an event type of USER and a severity of INFO.

**EDR_USER_WARNING_INJECT (trace, msg)**

Creates a record in the error log with event type of USER and a severity of WARNING.

**EDR_USER_FATAL_INJECT (trace, msg)**

Creates a record in the error log with event type of USER and a severity of FATAL.

All the macros use the same parameters. The *trace* parameter is a boolean value indicating whether or not a traceback should be generated for the record. The *msg* parameter is a string that is added to the record.

8.8 Sample Error Record

The following is an example of a record generated by a failed process task:
Severity/Facility: FATAL/RTP
Boot Cycle: 1
OS Version: 6.0.0
Time: THU JAN 01 05:21:16 1970 (ticks = 1156617)
Task: "tInitTask" (0x006f4010)
RTP: "edrdemo.vxe" (0x00634048)
RTP Address Space: 0x10226000 -> 0x10254000
Injection Point: rtpSigLib.c:4893

Default Signal Handling: Abnormal termination of RTP edrdemo.vxe (0x634048)

<<<Memory Map>>>>
0x00100000 -> 0x002a48dc: kernel
0x010226000 -> 0x010254000: RTP

<<<Registers>>>>

r0 = 0x10226210 sp  = 0x102242f70 r2 = 0x10238e30
r3 = 0x00000037 r4  = 0x1022440e8 r5 = 0x10244128
r6 = 0x00000000 r7  = 0x10231314 r8 = 0x00000000
r9 = 0x10226275 r10 = 0x0000000c r11 = 0x0000000c
r12 = 0x00000000 r13 = 0x10238e30 r14 = 0x00000000
r15 = 0x00000000 r16 = 0x10238e30 r17 = 0x00000000
r18 = 0x00000000 r19 = 0x10238e30 r20 = 0x00000000
r21 = 0x00000000 r22 = 0x10238e30 r23 = 0x00000000
r24 = 0x00000000 r25 = 0x10238e30 r26 = 0x00000000
r27 = 0x00000000 r28 = 0x10238e30 r29 = 0x00000000
r30 = 0x10238e30 r31 = 0x10238e30 mstatus = 0x00000000
r32 = 0x10238e30 r33 = 0x10238e30 pc = 0x10226214
r34 = 0x10238e30 r35 = 0x10238e30 r36 = 0x10238e30 pgTbPtr = 0x00074000
scSrTblPtr = 0x0064ad04 rSrTblPtr = 0x0064acc4

<<<Disassembly>>>>

0x102261f4  48003559    bl          0x1022974c # strtoul
0x102261f8  3be30000    addi        r31,r3,0x0 # 0
0x102261fc  38636244    addi        r3,r3,0x6244 # 25156
0x10226200  389f0000    addi        r4,r3,0x0 # 0
0x10226204  4cc63182    crxor       crb6,crb6,crb6
0x10226208  48000014    b           0x1022622c # 0x1022622c
0x10226212  39800000    li          r12,0x0 # 0
0x10226216  80010014    lwz         r0,20(r1)
0x1022621a  83e1000c    lwz         r31,12(r1)

<<<Traceback>>>>

0x102261cc _start +0x4c : main ()
9

C++ Development

9.1 Introduction 341
9.2 C++ Code Requirements 342
9.3 C++ Compiler Differences 342
9.4 Namespaces 345
9.5 C++ Demo Example 346

9.1 Introduction

This chapter provides information about C++ development for VxWorks using the Wind River and GNU toolchains.

⚠️ WARNING: Wind River Compiler C++ and GNU C++ binary files are not compatible.

⚠️ NOTE: This chapter provides information about facilities available for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the VxWorks Kernel Programmer’s Guide.
9.2 C++ Code Requirements

Any VxWorks task that uses C++ must be spawned with the VX_FP_TASK option. By default, tasks spawned from host tools (such as the Wind Shell) automatically have VX_FP_TASK enabled.

⚠️ **WARNING:** Failure to use the VX_FP_TASK option when spawning a task that uses C++ can result in hard-to-debug, unpredictable floating-point register corruption at run-time.

If you reference a (non-overloaded, global) C++ symbol from your C code you must give it C linkage by prototyping it using `extern "C"`:

```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.

9.3 C++ Compiler Differences

The Wind River C++ Compiler uses the Edison Design Group (EDG) C++ front end. It fully complies with the ANSI C++ Standard. For complete documentation on the Wind River Compiler and associated tools, see the Wind River C/C++ Compiler User’s Guide.

The GNU compilers provided with the host tools and Workbench 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. For complete documentation on the GNU compiler and on the associated tools, see the GNU ToolKit User’s Guide.

The following sections briefly describe the differences in compiler support for template instantiation, exception handling, and run-time type information.
9.3.1 Template Instantiation

In C, every function and variable used by a program must be defined in exactly one place (more precisely one translation unit). However, in C++ there are entities which have no clear point of definition but for which a definition is nevertheless required. These include template specializations (specific instances of a generic template; for example, `std::vector<int>`), out-of-line bodies for inline functions, and virtual function tables for classes without a non-inline virtual function. For such entities a source code definition typically appears in a header file and is included in multiple translation units.

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

It is highly recommended that you use the default settings for template instantiation, since these combine ease-of-use with minimal code size. However it is possible to change the template instantiation algorithm; see the compiler documentation for details.

**Wind River Compiler**

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

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

- `-Xcomdat-off`
  Generate template instantiations and inline functions as static entities in the resulting object file. Can result in multiple instances of static member-function or class variables.

For greater control of template instantiation, the `-Ximplicit-templates-off` option tells the compiler to instantiate templates only where explicitly called for in source code; for example:

```cpp
template class A<int>; // Instantiate A<int> and all member functions.
template int f1(int);  // Instantiate function int f1(int).
```
GNU Compiler

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

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

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

9.3.2 Exception Handling

Both compilers support thread-safe exception handling by default.

Wind River Compiler

To turn off support for exception handling, use the -Xexceptions-off compiler flag. The Wind River Compiler 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.

GNU Compiler

To turn off support for exception handling, use the -fno-exceptions compiler flag.

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.
9.3.3 Run-Time Type Information

Both compilers support Run-time Type Information (RTTI), and the feature is enabled by default. This feature adds a small overhead to any C++ program containing classes with virtual functions.

For the Wind River Compiler, the RTTI language feature can be disabled with the -Xrtti-off flag.

For the GNU compiler, the RTTI language feature can be disabled with the -fno-rtti flag.

9.4 Namespaces

Both the Wind River and GNU C++ compilers supports namespaces. You can use namespaces for your own code, according to the C++ standard.

The C++ standard also defines names from system header files in a namespace called std. The standard requires that you specify which names in a standard header file you will be using.

The following code is technically invalid under the latest standard, and will not work with this release. It compiled with a previous release of the GNU compiler, but will not compile under the current releases of either the Wind River or GNU C++ compilers:

```cpp
#include <iostream.h>
int main()
{
    cout << "Hello, world!" << endl;
}
```

The following examples provide three correct alternatives illustrating how the C++ standard would now represent this code. The examples compile with either the Wind River or the GNU C++ compiler:

```cpp
// 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;
}

### 9.5 C++ Demo Example

For a sample C++ application, see 
`installDir/vxworks-6.x/target/usr/apps/samples/cplusplus/factory`. 
A
access routines (POSIX) 184
aio_cancel() 278
aio_error() 279
testing completion 280
aio_fsync() 278
aio_read() 278
aio_return() 279
aio_suspend() 278
testing completion 280
aio_write() 278
aiocb, see control block (AIO)
ANSIC
function prototypes 20
header files 21
stdio package 274
application libraries 27
applications
APIs 25
building 24
development 18
executing 63
library routines 25
main() routine 23
ROMFS, bundling applications with 74
starting with rtpSpawn() 23
structure 19
system calls 25

B
backspace character, see delete character
binary semaphores 111
block devices
file systems, and 292–328
naming 262
building
applications 24

VxWorks component requirements 23, 24
archive file attribute (dosFs) 309
asynchronous I/O (POSIX) 277
see also control block (AIO)
see online aioPxLib
cancelling operations 279
control block 278
multiple requests, submitting 279
retrieving operation status 279
routines 277
attribute (POSIX)
prioceiling attribute 195
protocol attribute 194
attributes (POSIX) 184
specifying 185
C

C library 51
C++ development
  C and C++, referencing symbols between 342
  exception handling 344
  Run-Time Type Information (RTTI) 345
C++ support 341–346
  see also iostreams (C++)
cancelling threads (POSIX) 190
CD-ROM devices 319
cdromFs file systems 319
  see online cdromFsLib
character devices
  naming 262
characters, control (CTRL+x)
  tty 282
client-server communications 124
CLOCK_MONOTONIC 177
CLOCK_REALTIME 177
clocks
  see also system clock; clockLib(1)
  monotonic 177
  POSIX 177–180
  real-time 177
  system 96
close() using 269
closedir() 301, 308
clusters
  cluster groups 311
  disk space, allocating (dosFs) 310
    absolutely contiguous 311
    methods 311
    nearly contiguous 311
    single cluster 310
    extents 311
code
  pure 101
  shared 100
code examples
  asynchronous I/O completion, determining
    signals, using 280
  dosFs file systems
    file attributes, setting 310
  maximum contiguous areas, finding 314
  message queues
    attributes, examining (POSIX) 221, 222
    POSIX 224
    VxWorks 123
  mutual exclusion 112
  semaphores
    binary 112
    named 215
    recursive 117
    unnamed (POSIX) 211
tasks
  deleting safely 95
  round-robin time slice (POSIX) 207
  synchronization 113
threads
  creating, with attributes 185–186
COMP 134
components
  application requirements 23, 24
configuration
  event 127
  signals 151
configuration and build
  components 3
  tools 3
configuring
  dosFs file systems 304
  HRFS file systems 297
  TSFS 327
contexts
  task 78
CONTIG_MAX 312
control block (AIO) 278
  fields 278
control characters (CTRL+x)
  tty 282
conventions
  device naming 262
  file naming 262
  task names 91
  counting semaphores 117, 209
creat() 269
CTRL+C kernel shell abort) 283
CTRL+D (end-of-file) 282
CTRL+H
  delete character
    tty 282
CTRL+Q (resume)
    tty 282
CTRL+S (suspend)
    tty 282
CTRL+U (delete line)
    tty 282
CTRL+X (reboot)
    tty 283

D

data structures, shared 107–108
devices
  see also block devices; character devices; direct-access devices; drivers and specific device types
  accessing 261
  creating
    pipes 284
default 262
dosFs 262
  naming 262
  network 285
  NFS 285
  non-NFS 286
  pipes 283
  pseudo-memory 284
  serial I/O (terminal and pseudo-terminal) 280
  sockets 287
  working with, in VxWorks 280–287
  disks
    changing
      dosFs file systems 306

file systems, and 292–328
  mounting volumes 317
  organization (rawFs) 317
  synchronizing
    dosFs file systems 306
delayed tasks 80
delayed-suspended tasks 80
delete character (CTRL+H)
    tty 282
delete-line character (CTRL+U)
    tty 282
displaying information
  disk volume configuration, about 306
  DLL, see plug-ins
documentation 2
dosFs file systems 306
  see also block devices; CBIO interface; clusters; FAT tables
  see online dosFsLib
code examples
  file attributes, setting 310
  maximum contiguous area on devices, finding the 314
  configuring 304
devices, naming 262
directories, reading 308
disk space, allocating 310
  methods 311
disk volume
  configuration data, displaying 306
disks, changing 306
  file attributes 308
  ioctl( ) requests, supported 302, 314
  open( ), creating files with 268
  starting I/O 308
  subdirectories
    creating 307
    removing 307
  synchronizing volumes 306
dosFsFmtLib 304
dosFsLib 304
dosFsShow( ) 306
drivers 261

see also devices and specific driver types

file systems, and 292–328

memory 284

NFS 285

non-NFS network 286

pipe 283

pty (pseudo-terminal) 280
tty (terminal) 280

VxWorks, available in 280

DSI 135

routines 131

sending 128

task events register 131

eventSend() 131, 132

exception handling 99–100

C++ 344

signal handlers 100

executing

applications 63

exit() 94

E

ED&R, see error detection and reporting 329

end-of-file character (CTRL+D) 282

environment variables 17

errno 98–99

return values 99

error

memory error detection 248

error detection and reporting 329

APIs for application code 339

error records 332

fatal error handling options 335

persistent memory region 331

error handling options 335

error records 332

error status values 98–99

ersors

run-time error checking (RTEC) 255

eventClear() 131, 132

eventReceive() 131, 132

events 126

accessing event flags 130

and object deletion 129

and show routines 132

and task deletion 130

configuring 127

defined 126

receiving 127

from message queues 128

from semaphores 127

from tasks and ISRs 127

F

fclose() 276

fd, see file descriptors

FD_CLR 273

FD_ISSET 273

FD_SET 273

FD_ZERO 273

fdopen() 275

dxprintf() 276

FIFO

message queues, VxWorks 121

file descriptor table 264

file descriptors (fd) 264

see also files

see online ioLib

pending on multiple (select facility) 272

reclaiming 264

 redirection 265

standard input/output/error 265

file pointers (fp) 275

file system monitor 295

file systems

see also ROMFS file system; dosFs file systems;

TRFS file system; rawFs file systems;
tapeFs file systems; Target Server File System (TSFS); TrueFFS flash file systems

block devices, and 292–328

drivers, and 292–328

files

attributes (dosFs) 308

closing 269
contiguous (dosFs)
    absolutely 311
    nearly 311
creating 269
deleting 270
exporting to remote machines 285
hidden (dosFs) 309
I/O system, and 261
naming 262
opening 267
reading from 270
    remote machines, on 285
read-write (dosFs) 309
system (dosFs) 309
truncating 271
write-only (dosFs) 309
writing to 270
-fimplicit-templates compiler option 344
FIOATTRIBSET 309
FIOLONG 315
FIODISKCHANGE 318
FIODISKFORMAT 318
FIODELLNAME 302, 315, 318
    pipes, using with 284
FIOFSTATGET 302, 315
    FTP or RSH, using with 287
    NFS client devices, using with 286
FIOGETNAME 303, 315
    FTP or RSH, using with 287
    NFS client devices, using with 286
    pipes, using with 284
FIOLABELGET 315
FIOLABELSET 315
FIOMKDIR 307
FIOMOVE 303, 315
FIONCONTIG 315
FIONFREE 303, 315
FIOMSGS 284
FIONREAD 303, 315
    FTP or RSH, using with 287
    NFS client devices, using with 286
    pipes, using with 284
FIOREADDIR 303, 315
    FTP or RSH, using with 287
    NFS client devices, using with 286
    FTP or RSH, using with 287
    memory drivers, using with 285
    NFS client devices, using with 286
FIOSEXCEPT 281
FIOSECTOPT 307
    tty options, setting 281
FIOSYNC 284
    FTP or RSH, using with 287
    NFS client devices, using with 286
FIOTRUNC 311
FIOTWHERE 303, 316
    FTP or RSH, using with 287
    memory drivers, using with 285
    NFS client devices, using with 286
    floating-point support
    task options 93
flow-control characters (CTRL+Q and S)
    tty 282
    -fno-exceptions compiler option (C++) 344
    -fno-implicit-templates compiler option 344
    -fno-rtti compiler option (C++) 345
fopen( ) 275
fread( ) 275
fstat( ) 301, 308
FSTAT_DIR 307
FTP (File Transfer Protocol)
    ioctl functions, and 287
    fwrite( ) 271, 311
fwrite( ) 275
getc( ) 275
global variables 102
header files 20
    ANSI 21
        function prototypes 20
hiding internal details 22
nested 21
private 22
searching for 21
hidden files (dosFs) 309
Highly Reliable File System 297
hook routines 25
hooks, task
  routines callable by 98
HRFS file systems
  configuring 297
directories, reading 301
starting I/O 301
subdirectories
  removing 301
HRFS, see Highly Reliable File System 297

I
-I compiler option 21
I/O system
  see also I/O, asynchronous I/O 277
include files
  see also header files
INCLUDE_ATA
  configuring dosFs file systems 297, 305
INCLUDE_CDROMFS 321
INCLUDE_DISK_UTIL 305
INCLUDE_DOSFS 304
INCLUDE_DOSFS_CHKDSK 305
INCLUDE_DOSFS_DIR_FIXED 305
INCLUDE_DOSFS_DIR_VFAT 305
INCLUDE_DOSFS_FAT 305
INCLUDE_DOSFS_FMT 305
INCLUDE_DOSFS_MAIN 305
INCLUDE_POSIX_FTRUNCATE 271
INCLUDE_POSIX_MEM 182
INCLUDE_POSIX_SCHED 206
INCLUDE_POSIX_SEM 209
INCLUDE_RAWFS 317
INCLUDE_SIGNALS 151
INCLUDE_TAR 305
INCLUDE_VXEVENTS 127
INCLUDE_WDB_TSFS 327

INCLUDE_XBD 305
INCLUDE_XBD_PARTLIB 298, 306
INCLUDE_XBD_RAMDISK 298, 306
INCLUDE_XBD_TRANS 306
instantiation, template (C++) 344
interrupt service routines (ISR)
  and signals 152
interruptible
  message queue 123
  semaphore 120
intertask communications 105–151
  network 149
I/O system 260
  asynchronous I/O 277
  basic I/O (ioLib) 263
  buffered I/O 274
  control functions (ioctl( )) 271
  memory, accessing 284
  redirection 265
  serial devices 280
  stdio package (ansiStdio) 274
ioctl( ) 271
dosFs file system support 302, 314
  functions
    FTP, using with 287
    memory drivers, using with 285
    NFS client devices, using with 286
    pipes, using with 284
    RSH, using with 287
  non-NFS devices 287
  raw file system support 318
  tty options, setting 281
ioctlPathGet() 262
ioctlPathSet() 262

K
kernel
  and multitasking 78
  POSIX and VxWorks features, comparison of
    message queues 220
  priority levels 83
kernel shell
  aborting (CTRL+C)
tty 283
keyboard shortcuts 86
keyboard shortcuts
tty characters 282
kill() 152, 153

L
latency
preemptive locks 108
libc 51
libraries
application 27
shared 32
line mode (tty devices) 282
selecting 281
lio_listio() 278
locking
page (POSIX) 182
semaphores 208
task preemptive locks 84, 108
longjmp() 100

message channels 133
message queue
interruptible 123
message queues 120
see also msgQLib(1)
and VxWorks events 125
client-server example 124
displaying attributes 124
POSIX 218
see also mqPxLib(1)
attributes 220, 222
code examples
attributes, examining 221, 222
communicating by message queue 224, 226
notifying tasks 226
unlinking 223
VxWorks facilities, differences from 220
priority setting 122
queuing 124
VxWorks 121
code example 123
creating 122
deleting 122
queueing order 121
receiving messages 122
sending messages 122
timing out 122
waiting tasks 122
mlock() 182
mlockall() 182
mmanPxLib 182
mounting volumes
rawFs file systems 317
mq_close() 218, 223
mq_getattr() 218, 220
mq_notify() 218, 226
mq_open() 218, 222
mq_receive() 218, 223
mq_send() 218, 223
mq_setattr() 218, 220
mq_unlink() 218, 223
mqPxLib 218
msgQCreate() 122
msgQDelete() 122

memory
driver (memDrv) 284
error detection 248
locking (POSIX) 182
see also mmanPxLib(1)
management, seeming
memory management
paging (POSIX) 182
persistent memory region 331
pool 102
pseudo-I/O devices 284
swapping (POSIX) 182
memory management
error detection 248
heap and partition 246
memory management
component requirements 246
msgQEvStart() 131
msgQEvStop() 131
msgQReceive() 122
msgQSend() 122
msgQSend() 132
multitasking 78, 100
example 104
munlock() 182
munlockall() 182
mutexes (POSIX) 192
mutual exclusion 108–109
see also semLib(1)
code example 112
counting semaphores 117
preemptive locks 108
and reentrancy 102
VxWorks semaphores 113
binary 112
deletion safety 115
priority inheritance 114
priority inversion 114
recursive use 116

named semaphores (POSIX) 208
using 213
nanosleep() 96
using 180
netDrv
compared with TSFS 327
netDrv driver 286
network devices
see also FTP; NFS; RSH
NFS 285
non-NFS 286
Network File System, see NFS
networks
intertask communications 149
transparency 285
NFS (Network File System) devices 285
naming 262
open(), creating files with 268
ioctl functions, and 286
transparency 285
nfsDrv driver 285
NUM_RAWFS_FILES 317

O
O_CREAT 307
O_NONBLOCK 220
O_CREAT 213
O_EXCL 213
O_NONBLOCK 223
open() 267
access flags 267
files asynchronously, accessing 277
files with, creating 268
subdirectories, creating 307
opendir() 301, 308
operating system 182
OPT_7_BIT 281
OPT_ABORT 281
OPT_CRMOD 281
OPT_ECHO 281
OPT_LINE 281
OPT_MON_TRAP 281
OPT_RAW 281
OPT_TANDEM 281
OPT_TERMINAL 281

P
page locking 182
see also mmannPxLib(1)
paging 182
pended tasks 80
pended-suspended tasks 80
persistent memory region 331
pipeDevCreate() 125
pipes 125–126
see online pipeDrv
ioctl functions, and 284
select(), using with 126
Index

plug-ins 52
POSIX
  see also asynchronous I/O
  asynchronous I/O 277
clocks 177–180
  see also clockLib(1)
file truncation 271
memory-locking interface 182
message queues 218
  see also message queues; mqPxLib(1)
mutex attributes
  prioceiling attribute 195
  protocol attribute 194
page locking 182
  see also mmanPxLib(1)
paging 182
priority limits, getting task 207
priority numbering 199
scheduling 206
  see also scheduling; schedPxLib(1)
semaphores 208
  see also semaphores; semPxLib(1)
signal functions 153
  see also signals; sigLib(1)
swapping 182
thread attributes 184–186
  specifying 185
threads 183
timers 177–180
  see also timerLib(1)
posixPriorityNumbering global variable 199
preemptive locks 84, 108
preemptive priority scheduling 85
printErr( ) 276
prioceiling attribute 195
priority
  inheritance 114
  inversion 114
  message queues 122
  numbering 199
  preemptive, scheduling 85
task, setting
  VxWorks 83
processes
  application development 18
  configuring VxWorks for 8
  definition 10, 12
  environment variables 17
  inheritance and resource reclamation 16
  initial task 15
  life cycle 12
  memory and 14
  POSIX and 18, 199
  real time 6
  tasks and 15
  protocol attribute 194
PTHREAD 194
pthread_attr_getdetachstate( ) 188
pthread_attr_getinherit( ) 188
pthread_attr_getschedparam() 189
pthread_attr_getscope( ) 188
pthread_attr_getstackaddr() 188
pthread_attr_getstacksize() 188
pthread_attr_setdetachstate( ) 188
pthread_attr_setinherit( ) 188
pthread_attr_setschedparam() 189
pthread_attr_setschedparam() 189
pthread_attr_setscope( ) 188
pthread_attr_setstackaddr() 188
pthread_attr_setstacksize() 188
pthread_attr_t 184
pthread_cancel( ) 191
pthread_cleanup_pop( ) 191
pthread_cleanup_push( ) 191
pthread_getspecific() 189
pthread_key_create( ) 189
pthread_key_delete( ) 189
pthread_mutex_getprioceiling() 195
pthread_mutex_setprioceiling() 195
pthread_mutexattr_getprioceiling() 195
pthread_mutexattr_getprotocol() 194
pthread_mutexattr_setprioceiling() 195
pthread_mutexattr_setprotocol() 194
PTHREAD_PRIO_INHERIT 194
PTHREAD_PRIO_NONE 194
PTHREAD_PRIO_PROTECT 194
pthread_setcancelstate( ) 191
pthread_setcanceltype( ) 191
pthread_setspecific() 189
pthread_testcancel( ) 191
pty devices 280
see online ptyDrv

public objects
  tasks  91
pure code  101
putc()  275

Q
queuedsignals  153
queues
  see also message queues
    ordering (FIFO vs. priority)  119
    semaphore wait  119
queuing
    message queues  124

R
-R option (TSFS)  328
raise()  152
raw mode (tty devices)  282
rawFs file systems  316–319
  see online rawFsLib
disk organization  317
disk volume, mounting  317
  ioctl() requests, support for  318
  starting I/O  318
read()  270
readdir()  301, 308
ready tasks  80
real-time processes, see processes  6
reboot character (CTRL+X)
  tty  283
redirect
  265
reentrancy  100–104
regions, shared data  60
remove()  270
  subdirectories, removing  301, 307
resource reclamation
  processes  16
restart character (CTRL+C)
  tty  283
resume character (CTRL+Q)
  tty  282
rewinddir()  301, 308
ROM monitor trap (CTRL+X)
  tty  283
ROMFS
  bundling applications  74
ROMFS file system  323
round-robin scheduling
  defined  85
routines
  hook  25
  scheduling, for  206
RSH (Remote Shell protocol)
  ioctl functions, and  287
RTEC, seerun-time error checking  255
RTP, see processes
rtpSpawn()  23
running
  applications  63
run-time error checking (RTEC)  255
Run-Time Type Information (RTTI)  345
-RW option (TSFS)  328

S
SAL  142
sched_get_priority_max()  206
sched_get_priority_max()  207
sched_get_priority_min()  206
sched_get_priority_min()  207
sched_rr_get_interval()  206
schedPxLib  199, 206
schedulers  83
scheduling  83
  POSIX  206
    see also schedPxLib(1)
    algorithms  199
    priority limits  207
    priority numbering  199
    routines for  206
VxWorks
  preemptive locks  84, 108
  preemptive priority  85
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>round-robin</td>
<td>85</td>
</tr>
<tr>
<td>security</td>
<td></td>
</tr>
<tr>
<td>TSFS</td>
<td>327</td>
</tr>
<tr>
<td>select facility</td>
<td>272</td>
</tr>
<tr>
<td>see online selectLib</td>
<td></td>
</tr>
<tr>
<td>macros</td>
<td>273</td>
</tr>
<tr>
<td>select()</td>
<td></td>
</tr>
<tr>
<td>and pipes</td>
<td>126</td>
</tr>
<tr>
<td>select()</td>
<td>272</td>
</tr>
<tr>
<td>sem_close()</td>
<td>208, 214</td>
</tr>
<tr>
<td>SEM_DELETE_SAFE</td>
<td>115</td>
</tr>
<tr>
<td>sem_destroy()</td>
<td>208</td>
</tr>
<tr>
<td>sem_getvalue()</td>
<td>208</td>
</tr>
<tr>
<td>sem_init()</td>
<td>208, 210</td>
</tr>
<tr>
<td>SEM_INVERSION_SAFE</td>
<td>114</td>
</tr>
<tr>
<td>sem_open()</td>
<td>208, 213</td>
</tr>
<tr>
<td>sem_post()</td>
<td>208</td>
</tr>
<tr>
<td>sem_trywait()</td>
<td>208</td>
</tr>
<tr>
<td>sem_unlink()</td>
<td>208, 214</td>
</tr>
<tr>
<td>sem_wait()</td>
<td>208</td>
</tr>
<tr>
<td>semaphores</td>
<td>109</td>
</tr>
<tr>
<td>and VxWorks events</td>
<td>120</td>
</tr>
<tr>
<td>see also semLib(1)</td>
<td></td>
</tr>
<tr>
<td>counting</td>
<td>117</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>112, 113</td>
</tr>
<tr>
<td>queueing</td>
<td>119</td>
</tr>
<tr>
<td>synchronization</td>
<td>113</td>
</tr>
<tr>
<td>timing out</td>
<td>118</td>
</tr>
<tr>
<td>semBCreate()</td>
<td>110</td>
</tr>
<tr>
<td>semCCreate()</td>
<td>110</td>
</tr>
<tr>
<td>semDelete()</td>
<td>110</td>
</tr>
<tr>
<td>semEvStart()</td>
<td>131</td>
</tr>
<tr>
<td>semEvStop()</td>
<td>131</td>
</tr>
<tr>
<td>semFlush()</td>
<td>110, 113</td>
</tr>
<tr>
<td>semGive()</td>
<td>110</td>
</tr>
<tr>
<td>semGive()</td>
<td>132</td>
</tr>
<tr>
<td>semMCreate()</td>
<td>110</td>
</tr>
<tr>
<td>semPxLib</td>
<td>208</td>
</tr>
<tr>
<td>semPxLibInit()</td>
<td>209</td>
</tr>
<tr>
<td>semTake()</td>
<td>110</td>
</tr>
<tr>
<td>serial drivers</td>
<td>280</td>
</tr>
<tr>
<td>set_terminate() (C++)</td>
<td>344</td>
</tr>
<tr>
<td>setjmp()</td>
<td>100</td>
</tr>
<tr>
<td>shared code</td>
<td>100</td>
</tr>
<tr>
<td>shared data regions</td>
<td>60</td>
</tr>
<tr>
<td>shared data structures</td>
<td>107–108</td>
</tr>
<tr>
<td>shared libraries</td>
<td>32</td>
</tr>
<tr>
<td>shared memory</td>
<td></td>
</tr>
<tr>
<td>see also shared data regions</td>
<td>60</td>
</tr>
<tr>
<td>show()</td>
<td>124, 213</td>
</tr>
<tr>
<td>sigaction()</td>
<td>152</td>
</tr>
<tr>
<td>sigaddset()</td>
<td>152</td>
</tr>
<tr>
<td>sigdelset()</td>
<td>152</td>
</tr>
<tr>
<td>sigemptyset()</td>
<td>152</td>
</tr>
<tr>
<td>sigfillset()</td>
<td>152</td>
</tr>
<tr>
<td>sigInit()</td>
<td>151</td>
</tr>
<tr>
<td>sigismember()</td>
<td>152</td>
</tr>
<tr>
<td>signal handlers</td>
<td>152</td>
</tr>
<tr>
<td>signals</td>
<td>149, 149–151</td>
</tr>
<tr>
<td>see also sigLib(1)</td>
<td></td>
</tr>
<tr>
<td>configuring</td>
<td>151</td>
</tr>
<tr>
<td>and interrupt service routines</td>
<td>152</td>
</tr>
<tr>
<td>POSIX</td>
<td>153</td>
</tr>
<tr>
<td>queued</td>
<td>153</td>
</tr>
<tr>
<td>signal handlers</td>
<td>152</td>
</tr>
<tr>
<td>sigpending()</td>
<td>152</td>
</tr>
<tr>
<td>sigprocmask()</td>
<td>152</td>
</tr>
</tbody>
</table>

357
sigqueue( ) 153
sigsuspend( ) 152
sigtimedwait( ) 153
sigvec( ) 152
sigwaitinfo( ) 153
SNS 138
socket( ) 287
sockets
  I/O devices, as 287
  TSFS 326
spawning tasks 88–89, 103–104
stacks
  no fill 93
standard input/output/error
  basic I/O 265
  buffered I/O (ansiStdio) 276
starting
  applications 63
stat( ) 301, 308
stdio package
  ANSI C support 274
stopped tasks 80
subdirectories (dosFs)
  creating 307
  file attribute 309
suspended tasks 80
swapping 182
synchronization (task) 109
  code example 113
  counting semaphores, using 117
  semaphores 113
synchronizing media
  dosFs file systems 306
system calls 25
system clock 96
system files (dosFs) 309
task control blocks (TCB) 78, 97, 103
taskActivate( ) 89
taskCreate( ) 89
taskCreateHookAdd( ) 97
taskCreateHookDelete( ) 97
taskDelay( ) 96
taskDelete( ) 94
taskDeleteHookAdd( ) 97
taskDeleteHookDelete( ) 97
taskExit( ) 94
taskIdSelf( ) 92
taskIdVerify( ) 92
taskInfoGet( ) 94
taskIsPended( ) 94
taskIsReady( ) 94
taskIsSuspended( ) 94
taskName( ) 92
taskNameGet( ) 92
taskNameToId( ) 92
taskPriorityGet( ) 94
taskPrioritySet( ) 83, 84
taskRestart( ) 96
taskResume( ) 96
taskRtpLock( ) 83
taskRtpUnlock( ) 83
tasks
  blocked 84
  contexts 78
  control blocks 78, 97, 103
  creating 88–89
  delayed 80
  delayed-suspended 80
  delaying 79, 80, 96
  deleting safely 94–96
    code example 95
    semaphores, using 115
  displaying information about 93
  environment variables 17
  error status values 98–99
    see also errnoLib(1)
  exception handling 99–100
    see also signals; sigLib(1); excLib(1)
  executing 96
  hooks
    see also taskHookLib(1)

Target Server File System (TSFS) 325
configuring 327
error handling 327
file access permissions 327
sockets, working with 326

358
extending with 97–98
IDs 91
initial process task 15
names 91
  automatic 92
  private 91
  public 91
option parameters 92
pending 80
pended-suspended 80
priority, setting
  application tasks 83
  VxWorks 83
processes and 15
public 91
ready 80
scheduling
  POSIX 206
  preemptive locks 84, 108
  preemptive priority 85
  priority limits, getting 207
  round-robin 85
  VxWorks 83
shared code 100
and signals 100, 149–151
spawning 88–89, 103–104
stack allocation 89
stack protection 89
states 79–80
stopped 80
suspended 80
suspending and resuming 96
synchronization 109
  code example 113
  counting semaphores, using 117
task events register 131
  API 132
variables 102–103
  see also taskVarLib(1)
context switching 103
taskSafe() 94, 95
taskSpawn() 88
taskSuspend() 96
taskUnsafe() 95
taskVarAdd() 103
taskVarDelete() 103
taskVarGet() 103
taskVarSet() 103
terminate() (C++) 344
threads (POSIX) 183
  attributes 184–186
  specifying 185
  keys 189
  private data, accessing 189
  terminating 190
time slicing 86
timeout
  message queues 122
  semaphores 118
timeouts
  semaphores 118
timers
  see also timerLib(1)
  message queues, for (VxWorks) 122
  POSIX 177–180
  semaphores, for (VxWorks) 118
tools
  configuration and build 3
Transaction-Based Reliable File System, see TRFS 287
TRFS file system 287
truncation of files 271
tty devices 280
  see online tyLib
    control characters (CTRL+x) 282
    line mode 282
    options 281
      all, setting 281
      none, setting 281
    raw mode 282
    X-on/X-off 281
tyBackspaceSet() 283
tyDeleteLineSet() 283
tyEOFSet() 283
tyMonitorTrapSet() 283
U
unnamed semaphores (POSIX)  208, 209, 210–212
usrRtpAppInit( )  72

V
variables
  global  102
  static data  102
  task  102–103
Virtual Root File System  295
volume labels (dosFs)
  file attribute  309
VX_ALTIVEC_TASK  93
VX_DSP_TASK  93
VX_FP_TASK  93, 342
VX_FP_TASK option  93
VX_NO_STACK_FILL  93
VX_PRIVATE_ENV  93
VxWorks
  components  3
  configuration and build  3
  header files  20
  message queues  121
  real-time processes, see processes  6
VxWorks events, see events
VxWorks facilities
  POSIX, differences from
    message queues  220
    scheduling  83
vxWorks.h  20

W
WAIT FOREVER  119
write( )  270