

VxWorks® 653

PROGRAMMER'S GUIDE

2.2

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, the Wind River logo, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:
installDir\product_name\3rd_party_licensor_notice.pdf.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 501-1153
U.S.A.

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

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

<http://www.windriver.com>

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

<http://www.windriver.com/support>

Contents

1	Overview	1
1.1	About This Documentation	1
1.2	Overview of VxWorks 653	2
1.2.1	Overview of the vThreads Partition OS	3
1.2.2	Overview of COIL	4
1.3	Run-time System	4
1.3.1	Run-time Layers	5
	Core OS Layer	6
	vThreads Layer	7
	COIL Layer	7
	APEX Layer	8
	POSIX Layer	8
1.3.2	Loading and Booting	8
1.3.3	Run-time Model	9
1.4	RTCA/DO-178B Certifiability	9

2	Developing vThreads Applications	13
2.1	Introduction	13
2.2	vThreads Time Management	15
2.2.1	vThreads Timer Queue	15
2.2.2	vThreads Scheduling	16
	Priority-Preemptive Scheduling	17
	Round-Robin Scheduling	18
2.3	Handling External Stimuli	18
2.3.1	vThreads Pseudo-Interrupt Signals	19
	Pseudo-Interrupt Events Forbidden in User Handlers	21
	Pseudo-interrupt Events Permitted in User Handlers	22
2.3.2	vThreads Synchronous Exception Handling	23
2.4	vThreads Memory Management	24
2.5	vThreads Initialization and Restart	25
2.5.1	vThreads Boot Sequence	25
2.5.2	vThreads Restart	27
	Cold Versus Warm Restarts	27
	Cooperative Warm Partition Restart Mechanism	28
	Partition Restart and Device Drivers	30
2.6	Stack Overflow Protection	30
2.6.1	Guard Pages	31
2.6.2	Defaults	31
2.6.3	Limitations	32
2.7	vThreads Device I/O	33
2.8	vThreads APIs	33
2.9	vThreads System Calls	34

3	Developing COIL Applications	35
3.1	Introduction	35
3.2	VxWorks 653 Architecture and COIL	36
3.3	Accessing Core OS Services	37
3.4	Communicating with Other Partitions	37
3.5	Handling Interrupts and Exceptions	38
3.5.1	Handling Pseudo-Interrupts	39
3.5.2	Handling Exceptions	40
3.6	Restarting COIL Partitions	41
3.7	Device I/O in COIL Partitions	41
3.8	Monitoring Health in COIL Partitions	41
3.9	COIL API	41
4	Developing APEX Applications	43
4.1	Introduction	43
4.2	Adding APEX Support to vThreads Partitions	45
4.3	Terminology and Concepts: APEX Versus vThreads	45
4.4	Managing APEX Partitions	46
4.4.1	Allocating Partition Memory	46
4.4.2	Initializing Partitions: Cold and Warm Starts	47
4.4.3	Partition Attributes	47
4.4.4	Getting Partition Status	48
4.4.5	Setting the Partition Mode	49
4.4.6	Controlling Preemption in Partitions	50
4.4.7	Setting New Partition Schedules	51

4.5	Managing APEX Processes	51
4.5.1	Creating Processes	52
4.5.2	Changing the Current Priority of Processes	53
4.5.3	Increasing Deadline Times	53
4.5.4	Getting the Current Status of Processes	53
4.5.5	Getting Process IDs	54
4.5.6	Getting and Using vThreads Task Information	54
4.5.7	Types of Processes	54
4.5.8	Scheduling Processes	55
4.5.9	Process State Transitions	56
	DORMANT State	56
	WAITING State	57
	RUNNING State	58
	READY State	58
4.5.10	Suspending and Resuming Processes	59
4.5.11	Stopping and Starting Processes	60
4.5.12	Controlling Preemption	61
4.6	Managing Time in APEX Partitions	61
4.6.1	Scheduling Partitions	61
4.6.2	System Clock Time	61
4.6.3	Requesting Resources and Timeouts	61
4.6.4	Scheduling Processes	62
4.6.5	Deadlines	62
4.6.6	Release Points	66
4.7	Communicating between Partitions	67
4.7.1	Limitations of APEX for Communicating between Partitions	68
4.7.2	APEX Messages	68

4.7.3	APEX Channels	68
	Sampling Mode	69
	Queuing Mode	69
4.7.4	Ports	70
4.7.5	Working with Queuing Messages	72
4.7.6	Working with Sampling Messages	73
4.8	Communicating with Other Modules	74
4.8.1	Communicating Through Pseudo-Ports in a Pseudo-Partition	74
	Communicating Through Direct-Access Ports in a Pseudo-Partition	75
4.8.2	Communicating Through Direct-Access Ports in a Partition	76
	Sending and Receiving Messages	77
4.9	Communicating within APEX Partitions	77
4.9.1	Communicating Using APEX Buffers	77
4.9.2	Communicating Using APEX Blackboards	79
4.9.3	Communicating Using APEX Semaphores	82
4.9.4	Synchronizing Using APEX Events	83
4.10	Monitoring Health in APEX Partitions	86
4.10.1	Raising Process-Level Errors	86
4.10.2	APEX Errors	86
4.10.3	Creating Error Handler Processes	86
5	Developing POSIX Applications	89
5.1	Introduction	89
5.2	POSIX Clocks and Timers	90
5.3	POSIX Memory-Locking Interface	91

5.4	POSIX Threads	92
5.4.1	pThread Attributes	92
	Stack Size	92
	Stack Address	92
	Detach State	93
	Contention Scope	93
	Inherit Scheduling	94
	Scheduling Policy	94
	Scheduling Parameters	95
	Specifying Attributes when Creating pThreads	95
5.4.2	pThread Private Data	96
5.4.3	pThread Cancellation	97
5.5	POSIX Scheduling Interface	97
5.5.1	Comparison of POSIX and Wind Scheduling	98
5.5.2	Getting and Setting POSIX Task Priorities	98
5.5.3	Getting and Displaying the Current Scheduling Policy	100
5.5.4	Getting Scheduling Parameters: Priority Limits and Time Slice	101
5.6	POSIX Semaphores	101
5.6.1	Comparison of POSIX and Wind Semaphores	102
5.6.2	Using Unnamed Semaphores	103
5.6.3	Using Named Semaphores	105
5.7	POSIX Mutexes and Condition Variables	108
5.8	POSIX Message Queues	109
5.8.1	Comparison of POSIX and Wind Message Queues	109
5.8.2	POSIX Message Queue Attributes	110
5.8.3	Displaying Message-Queue Attributes	112
5.8.4	Communicating through a Message Queue	112
5.8.5	Notifying a Task That a Message Is Waiting	115

5.9	POSIX Queued Signals	120
5.10	POSIX API for vThreads Partitions	121
6	Developing C++ Applications	123
6.1	Introduction	123
6.2	Configuring vThreads to Use C++	124
6.2.1	Specifying Additional Sections for Loading	124
6.2.2	Adding C++ Support to vThreads	124
6.2.3	Demangling C++ Symbol Names in the Target Shell	124
6.3	Writing C++ Applications	124
6.3.1	Making C Symbols Accessible to C++ Code	125
	Making C++ Symbols Accessible to C code	125
6.3.2	Adding Floating-Point Support to Tasks	125
6.3.3	Handling Exceptions	125
	Turning off Exception Handling	125
	Using the Pre-Exception Model of C++ Compilation	125
	Installing Your Own Termination Handler	126
6.3.4	Using Namespaces	126
6.3.5	Disabling Run Time Type Information (RTTI)	126
6.3.6	Constructors and Destructors	127
6.4	Using C++ Libraries	127
6.4.1	Using the iostream Library	127
	Standard iostream Objects	127
6.4.2	Using Standard Template Library (STL)	128
6.5	Writing C++ Cert Applications	128
6.5.1	Features Not Supported	129

6.5.2	Persistent Global Constructors	129
	Specifying Persistent Global Constructors in Makefiles	129
	Allocating Persistent Global Constructors	129
6.5.3	Calling Pure Virtual Functions	130
6.5.4	Deallocating Heap	130
7	Programming in the Core OS	133
7.1	Introduction	134
7.2	Partitions	135
7.2.1	Partition Configuration	135
	System Call Permission Bitmasks	138
	PPS Scheduling Parameters	140
7.3	VxWorks 653 Stacks	141
	System Call Stacks	141
	Task Stacks	141
	Task Exception Stacks	141
	Interrupt Stack	142
7.4	Shared Libraries	142
7.4.1	Adding User-supplied Code to a Partition OS	143
7.5	Shared Data Regions	143
7.6	User Configuration Records	147
7.7	Multitasking	147
7.8	Managing Memory	147
7.8.1	Managing Memory Partitions and Heaps	148
	Managing Memory Partitions	148
	Managing Typed Memory Partitions	149
	Managing the Current Heap	149

7.8.2	Managing Virtual Memory	150
	Accessing the MMU	151
	Ensuring Cache Coherency	151
	Write-Protecting Text Segments	152
	Write-Protecting the Exception Vector Table	152
	Virtual Memory Contexts and Domains	152
7.8.3	Managing Page-oriented Memory	153
	Managing Physical Memory	153
	Managing Virtual Pages	153
7.8.4	POSIX Memory-Locking Interface	156
7.9	Restart Functionality	156
7.9.1	System Cold Start or Restart	157
7.9.2	System Warm Restart	159
	Including Warm Restart in a BSP	159
7.9.3	Partition Cold Start or Restart	160
7.9.4	Partition Warm Restart	161
7.9.5	Restart Implications for Drivers	162
7.9.6	Restart Implications for I/O	163
7.9.7	Persistent Data Support for Restart	163
7.10	Partition Support	165
7.10.1	Core OS Partition-Related Components	165
7.10.2	Core OS Partition-Related Routines	165
7.10.3	Online-Loaded Partitions	165
7.11	Worker Tasks	168
7.12	System Time	169
7.13	Partition Scheduling	169
7.13.1	TPS Scheduling	170
	Scheduling Rules	170
	Partition Activation	170

	Spare-Time Monitoring	171
	Mode-Based Scheduling	171
7.13.2	APPS Scheduling	172
	How the Kernel Identifies Idle Time	175
	vThreads and APPS Scheduling	176
	Ticks and Timeouts	176
	Pseudo-Interrupts	177
	Examples of APPS Scheduling	178
7.13.3	Partition-Scheduling Routines	181
7.14	Design Models for Ports	181
7.14.1	Design Model for Queuing Ports	181
	Memory Use	181
	Blocking Processes	182
	System Calls and Events for Port Operations	183
	Effect of Restarting Partitions	183
7.14.2	Design Model for Sampling Ports	183
7.15	Setting up Communication with Other Modules	185
7.15.1	Configuring a Supervisor-Level Driver	186
7.15.2	Adding a Driver	186
7.15.3	Driver Routines	186
	Attaching the Name of a Driver to a Pseudo-Port ID	187
	Reading Messages from a Pseudo-Port	187
	Writing Messages to a Pseudo-Port	188
	Determining the Availability of a Pseudo-Port	188
	Getting the Status of a Pseudo-Port	189
	Determining Whether a Pseudo-Port Is Direct Access	189
	Function Pointer Structure for Drivers	189
7.15.4	Sending and Receiving Messages	189
	Sending Messages	189
	Receiving Messages	190
	Time Partitioning	190
7.15.5	Example: Communicating between Modules	191
	Configuration of Module A	191
	Configuration of Module B	192

	User-Supplied Code for Module A's Send Operation	193
	User-Supplied Code for Module B's Receive Operation	194
8	Health Monitoring	195
8.1	Introduction	195
8.2	Basic Health Monitor Concepts	196
8.2.1	Health Monitor Events	196
	Health Monitor Alarms	196
	Health Monitor Messages	196
8.2.2	Health Monitor Hierarchy	197
8.2.3	Event Structure (HM_EVENT)	199
	System Status and Modes	200
8.2.4	Injecting Alarms	202
	Dispatching and Logging Messages	208
8.2.5	Health Monitor Thresholds	209
	Notification Queue Threshold	209
	Log Threshold	209
	Event Queue Threshold	209
	Error Handler Queue Threshold	210
8.3	Health Monitor Actions	210
8.3.1	Escalating Alarms	210
8.3.2	Logging Events	211
8.3.3	Notifying Other Partitions	212
8.3.4	Issuing Callbacks	212
8.3.5	Detecting and Reporting Application Errors	213
	Reporting for ARINC 653 Applications	213
	ARINC 653 Errors and Health Monitor Equivalents	214
	Reporting for Non-ARINC 653 Applications	215
8.4	Initializing the Health Monitor	215
8.5	Getting Health Monitor Information at Run-time	215

8.6	Defining the Health Monitor Handler Table	216
8.6.1	Guidelines for Writing Handlers	216
8.7	Health Monitoring for COIL Partitions	217
8.8	Other Facilities That Inject Alarms	218
8.9	Public Information	218
9	I/O Support	221
9.1	Introduction	221
9.2	I/O and vThreads	221
9.2.1	vThreads I/O and Worker Tasks	222
9.2.2	Device Driver Models	223
	vThreads Model of Device Drivers	224
	Core OS Model of Device Drivers	225
	Split Model of Device Drivers	226
9.2.3	Select Capability	228
	Supervisor-Level Device Driver Model	256
9.3	Application Multiplexed I/O	256
9.3.1	Serialized I/O Protocol	257
9.3.2	Architecture	258
9.3.3	Setting up and Using Application Multiplexed I/O in Partitions	260
	Making the Driver Available	260
	Redirecting Standard I/O to the pamio Driver	260
	Using Application Multiplexed I/O	260
9.3.4	Using Application Multiplexed I/O in the Core OS	261
	Setting the Mux/Demux Algorithm	261
	Using the ioctl() Routine	262
	Using the mamio Driver for All I/O in the Core OS	263

9.4	I/O and COIL	264
	Blocking Versus Non-blocking I/O (Compared to vThreads)	264
	Non-blocking COIL I/O (Worker Tasks Present)	266
	Blocking I/O (No Worker Tasks)	267
A	VxWorks 5.5	269
A.1	Introduction	269
A.2	VxWorks Tasks	270
A.2.1	Multitasking	270
A.2.2	Task State Transition	271
A.2.3	Wind Task Scheduling	273
	Priority-Preemptive Scheduling	273
	Round-Robin Scheduling	274
	Preemption Locks	275
	A Comparison of taskLock() and intLock()	276
	Driver Support Task Priority	276
A.2.4	Task Control	277
	Task Creation and Activation	277
	Task Stack	278
	Task Names and IDs	278
	Task Options	279
	Task Information	280
	Task Deletion and Deletion Safety	280
	Task Control	282
A.2.5	Tasking Extensions	283
A.2.6	Task Error Status: errno	285
	Layered Definitions of errno	285
	A Separate errno Value for Each Task	286
	Error Return Convention	286
	Assignment of Error Status Values	287
A.2.7	Task Exception Handling	287
A.2.8	Shared Code and Reentrancy	288
	Dynamic Stack Variables	289
	Guarded Global and Static Variables	290

	Task Variables	290
	Multiple Tasks with the Same Main Routine	292
A.2.9	VxWorks System Tasks	292
	Root Task: tUsrRoot	293
	Logging Task: tLogTask	293
	Exception Task: tExcTask	293
	Tasks for Optional Components	293
A.3	Intertask Communications	294
A.3.1	Shared Data Structures	294
A.3.2	Mutual Exclusion	295
	Interrupt Locks and Latency	295
	Preemptive Locks and Latency	296
A.3.3	Semaphores	296
	Semaphore Control	297
	Binary Semaphores	298
	Mutual-Exclusion Semaphores	302
	Counting Semaphores	305
	Special Semaphore Options	306
	Semaphores and VxWorks Events	307
A.3.4	Message Queues	309
	Wind Message Queues	310
	Displaying Message Queue Attributes	312
	Servers and Clients with Message Queues	312
	Message Queues and VxWorks Events	313
A.3.5	Pipes	315
A.3.6	Signals	315
	Basic Signal Routines	316
	Signal Configuration	316
A.4	VxWorks Events	317
A.4.1	Free Resource Definition	318
A.4.2	Single-Task Resource Registration	319
A.4.3	Option for Immediate Send	319
A.4.4	Option for Automatic Unregister	320

A.4.5	Automatic Unpend upon Resource Deletion	320
A.4.6	Task Events Register	320
A.4.7	VxWorks Events API	321
A.4.8	Show Routines	321
A.5	Watchdog Timers	322
A.6	Interrupt Service Routines	323
A.6.1	Interrupt Stack	324
A.6.2	Writing and Debugging ISRs	324
A.6.3	Special Limitations of ISRs	324
A.6.4	Exceptions at Interrupt Level	327
A.6.5	Reserving High Interrupt Levels	327
A.6.6	Additional Restrictions for ISRs at High Interrupt Levels	327
A.6.7	Interrupt-to-Task Communication	328
B	PowerPC Considerations	329
B.1	Introduction	329
B.2	Building Applications	330
	Defining the CPU-Type Configuration Variable (CPU)	330
	Setting Compiler Options	331
B.3	Memory Management Unit	332
B.3.1	Enabling or Disabling Instruction MMUs and Data MMUs	332
B.3.2	Mapping Memory (PowerPC 60x)	332
	BAT Model for Mapping Memory	332
	Segment Model for Mapping Memory	332
B.3.3	Setting MMU Access Rights	333
B.3.4	Setting MMU Cache Attributes	334
	Determining the Size of Hash Tables (PowerPC 604)	335
	Resizing and Moving Hash Tables (PowerPC 604)	336

B.3.5	ELF-Specific Tools	336
B.3.6	Detecting NULL Pointer Dereferences	337
B.4	Protection Domains (PowerPC 60x)	337
B.5	Architecture Considerations	337
	Processor Mode	338
	24-bit Addressing	338
	Byte Order	338
	PowerPC Registers	338
	HI and HIADJ Macros	339
	Cache and Kernel Heap	340
	Floating-Point Routines	340
	Support for Floating-Point Exceptions in Partitions	342
	Shared Library Support (PowerPC 604)	343
	Debugging	343
C	Glossary	347
	Index	371

1

Overview

- 1.1 About This Documentation 1
- 1.2 Overview of VxWorks 653 2
- 1.3 Run-time System 4
- 1.4 RTCA/DO-178B Certifiability 9

1.1 About This Documentation

This documentation describes the VxWorks 653 real-time operating system (RTOS) and how to use its run-time facilities to develop embedded, safety-critical applications and systems.

Cross-references to libraries, routines, commands, and utilities refer to reference entries in the API reference documentation that is available from the Wind River Workbench online help. To access the reference entry, click **Help > Search**, type the name of the entry in the **Search Expression** box, and click **Go**.

Alternatively, click **Help > Help Contents**. In the left pane of the help system that opens, click **Wind River Documentation > References**. Under **Operating System** and **Host Tools** are numerous API reference documentation.

For Information On:	See:
Configuring and building systems	<i>VxWorks 653 Configuration and Build Guide</i> <i>VxWorks 653 Configuration and Build Reference</i>
Loading, running, and debugging	<i>Wind River Workbench User's Guide (VxWorks 653 Version)</i>
Terms used in the documentation	C. Glossary

1.2 Overview of VxWorks 653

VxWorks 653 fully complies with the *Avionics Application Software Standard Interface*, ARINC 653, Supplement 2, Part 1 *Required Services*. VxWorks 653 does not support any Supplement 2, Part 1 optional features, all of which reduce the strength of the specification.

VxWorks 653 also supports standard service access point (SAP) ports, which are defined in ARINC 653, Supplement 2, Part 2 *Extended Services*.

Subsets of VxWorks 653 are available that can be certified to Level A of the RTCA/DO-178B avionics software guidelines. VxWorks 653 is, therefore, suitable for safety-critical applications.

A VxWorks 653 module is the system controlled by one RTOS, and that RTOS is the core OS of VxWorks 653. Unless it states otherwise, this documentation assumes you are working within one module.

Within a module, VxWorks 653 supports complete separation between applications and between applications and the module's core OS. As a result, applications can interact with each other only through explicit mechanisms that the core OS controls. Applications cannot affect the operation of the module, except in a controlled manner through resources that the core OS explicitly allocates to them.

Each application runs in a discrete partition. The core OS controls the partitions by providing time and space partitioning and memory management services. Partitions manage their own resources within the time slot that the core OS provides. Performance is optimized by keeping as many routine calls as possible within the partition. Partitions run in user mode. The core OS runs in supervisor mode.

Each partition contains a partition-level OS (the partition OS) with a set of OS services. VxWorks 653 provides the vThreads partition OS and COIL (a partition OS independent of vThreads). You can augment COIL to suit specific partition OS needs. For vThreads partitions, VxWorks 653 supports the POSIX and APEX interfaces.

VxWorks 653 supports warm start and cold start of partitions and of the entire module.

VxWorks 653 includes libraries for developing and debugging applications. Included is support for the Wind River System Viewer GUI-based software logic analyzer, back-ends for host-target communication, and a loader.

1.2.1 Overview of the vThreads Partition OS

The vThreads partition OS includes its own set of objects (for example threads, semaphores, and mutexes), other libraries, and internal scheduling. The core OS performs the following for vThreads:

- timer facility
- I/O operations
- some scheduling
- interpartition communication

vThreads can access only its own memory heap.

vThreads cannot directly access I/O devices or supervisor-level processor resources. The core OS provides these services through system calls.

vThreads does not directly receive hardware interrupts or exceptions. The core OS sends a pseudo-interrupt to the appropriate partition, and vThreads handles it as if it were the real interrupt.

A subset of vThreads is available that is certifiable to Level A of the RTCA/DO-178B avionics software guidelines.

For details, see [2. Developing vThreads Applications](#).

1.2.2 Overview of COIL

The core OS interface library (COIL) is a library that lets you implement a partition OS that is not based on vThreads. COIL includes the minimum services needed for an application to communicate with the core OS. These services include the following:

- interrupt and exception management
- device I/O
- interpartition messaging
- injection of health monitor events

COIL supports APPS scheduling.

A subset of COIL is available that can be certified to Level A of the RTCA/DO-178B avionics software guidelines.

For details, see [3. Developing COIL Applications](#).

1.3 Run-time System

Applications are compiled against the appropriate partition OS header files and are linked against the libraries available within the partition. At boot time, the core OS loads each application to its own partition.

The vThreads partition OS offers the vThreads API for C applications. Also, vThreads lets applications use either the Ada, APEX, or POSIX interfaces or use the C++ language. You can add application or third-party header files and libraries to the compiling and linking mechanisms for both the core OS and partition OSs.

Applications call routines located in their partition OS. The partition OS completes the routine autonomously if it provides the requested service. Otherwise, if the application's privileges permit, the partition OS makes a system call to the core OS. For example, a system call occurs when the I/O subsystem calls **read()** or when a message is sent to an application in another partition.

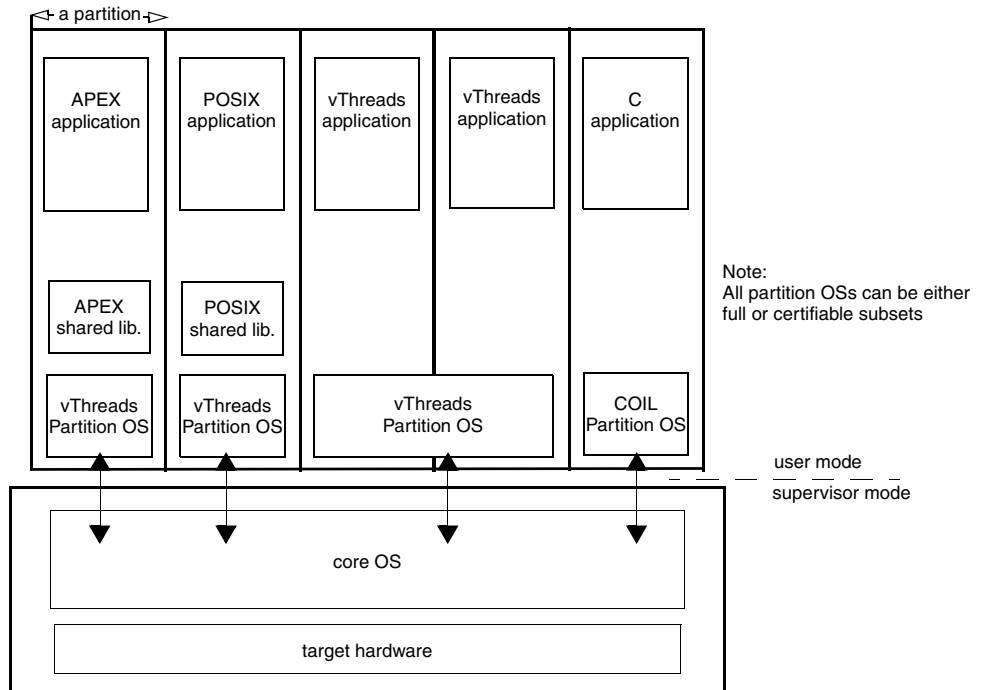
1.3.1 Run-time Layers

A VxWorks 653 module consists of up to four basic types of layers:

- core OS—Required.
- partition—At least one is required (vThreads or COIL-based), each in a partition OS.
- APEX shared library—Required for ARINC 653 applications.
- POSIX shared library—Required for POSIX applications.

Figure 1-1 shows a VxWorks 653 module with five partitions. In this example, the two vThreads partitions share the same partition OS. For shared libraries and partition OSs, the figure shows relative virtual addresses within a partition, not within the module.

Figure 1-1 A VxWorks 653 Module with Five Partitions



Core OS Layer

The core OS provides services to the partitions.

By default, the core OS schedules partitions by ARINC 653 (time-preemptive) scheduling (TPS) and in partition order. However, if a partition is enabled for priority-preemptive scheduling (PPS), the core OS considers the partition for APPS scheduling. APPS scheduling schedules PPS-enabled partitions during the idle time within a TPS schedule. For more information on scheduling partitions, see [7.13 Partition Scheduling](#), p.169.

Key Core OS Services for vThreads

The core OS does the following for each vThreads partition OS:

- Allocates system resources.
- Schedules partitions.
- Traps exceptions on behalf of the partition OS.
- Defines and enforces partition boundaries.
- Loads partitions.
- Passes messages between partitions using ports and channels
- Handles I/O.
- Performs system calls on behalf of the applications.
- Supports debugging of partitions.
- Monitors the health of partitions and the system.

Key Core OS Services for COIL

The core OS does the following for each COIL-based partition OS:

- Manages interrupts and exceptions by pseudo-interrupts.
- Provides I/O support for devices and ports.
- Provides interpartition messaging via ports.
- Monitors health at these levels: VxWorks 653 module, partition, and process.

Core OS Certifiability

You can use a core OS certifiable subset, whose features have been selected for predictability, functionality, and certifiability under Level A objectives of the RTCA/DO-178B avionics software guidelines.

The subset provides an operating environment that supports the development of safety-critical applications. In addition, the subset includes a set of debug libraries that supports minimal debugging over a serial connection. The libraries must be removed before the application is deployed.

vThreads Layer

The core OS does not schedule vThreads threads. The scheduler in the partition OS schedules them during the time that the core OS allocates to the partition.

vThreads does not directly interact with devices, but instead defers those operations to the core OS by making system calls. System calls are also made for the timer, some scheduling facilities, and interpartition communication.

For more information, see [2. Developing vThreads Applications](#) and [6. Developing C++ Applications](#).

vThreads Certifiability

You can use a vThreads certifiable subset, whose features have been selected for certifiability to Level A of the RTCA/DO-178B avionics software guidelines. The certifiable subset supports the functionality of the APEX layer, as described in the ARINC 653 specification.

The vThreads certifiable subset can be used with the core OS certifiable subset and a COIL certifiable subset.

COIL Layer

COIL provides an API for the minimum functionality for an application to communicate with the core OS: interrupt and exception management, device I/O, interpartition messaging, and the injection of health monitor events.

For more information, see [3. Developing COIL Applications](#).

COIL Certifiability

You can use a COIL certifiable subset, whose features have been selected for certifiability to Level A of the RTCA/DO-178B avionics software guidelines.

The COIL certifiable subset can be used with the core OS certifiable subset and a vThreads certifiable subset.

APEX Layer

The APEX layer is built on top of vThreads and conforms to the ARINC 653 specification for functionality and API. For details, see [4. Developing APEX Applications](#).

POSIX Layer

This POSIX layer is built on top of vThreads and conforms to the POSIX standard for real-time extensions (1003.1b). For details, see [5. Developing POSIX Applications](#).

1.3.2 Loading and Booting

When power is applied to the target, the following happens:

- The initial boot code loads the core OS, partition OSs, shared libraries, and applications.
- The core OS initializes itself, starting its own subsystems.
- The core OS creates the partitions.
- The core OS starts the partition scheduler, letting applications initialize themselves.

Because the core OS loads applications as part of the boot sequence, incremental loading is not available. However, the core OS can download online-loaded applications into partitions after initialization is complete. The application is identified as destined for online loading. The application and the rest of the VxWorks 653 module are built together. Then the application can be loaded into the partition after the partition is running. For more information, see [7.10.3 Online-Loaded Partitions](#), p.165.

Restart requires a mechanism for reloading partitions from flash on restart. This is implemented using payload images rather than a boot application or the standard

ROM-resident image. These images provide section information to the restart mechanism to load the images from either RAM or flash without the use of a boot application. For more information, see the *VxWorks 653 Configuration and Build Guide*.

1.3.3 Run-time Model

The core OS handles system calls from each partition and validates all arguments of each system call before running it. Applications that use the vThreads partition OS have the complete set of vThreads intertask communication mechanisms available to them for use within a partition.

In addition, applications that use APEX libraries to provide ARINC 653 support have additional capabilities. APEX provides partition management, process management, and time management that conform to the ARINC 653 specification. APEX provides messages, channels, and ports for interpartition communication, as well as buffers, blackboards, semaphores, and events for intra-partition communication. For more information, see [4.7 Communicating between Partitions](#), p.67 and [4.9 Communicating within APEX Partitions](#), p.77.

Port mapping allows communicating outside the VxWorks 653 module. For details, see [4.8 Communicating with Other Modules](#), p.74.

1.4 RTCA/DO-178B Certifiability

To support Level A certification to the RTCA/DO-178B avionics software guidelines, a certifiable subset of VxWorks 653 is available. The certifiable subset is selected to comply with the objectives of RTCA/DO-178B. Selection is based on the deterministic nature of the code. The subset excludes operations that can compromise the integrity of safety-critical systems (for example, dynamically deallocating memory).

After you develop, debug, and fine-tune an application on the full VxWorks 653 (by using the debug build spec), you can move it to the certifiable subset (by using the cert build spec).

Because VxWorks 653 configuration does not prevent you from including debugging components in the certifiable image, it is your responsibility to ensure all debugging components are removed before the application is deployed.

The following combinations of debug core OS, debug vThreads (or debug COIL), and their certifiable subsets are recommended:

	Debug vThreads Debug COIL	vThreads certifiable subset or COIL certifiable subset
Debug core OS	For development work.	For debugging certifiable applications.
Core OS certifiable subset	N/A	For deployed systems.

Recommendations to Ensure RTCA/DO-178B Level A Certifiability

The following recommendations are made to ensure VxWorks 653 is used in a manner consistent with the Level A objectives defined by RTCA/DO-178B:

- System objects and resources (for example memory, queues, tasks, and semaphores) must be allocated only when an application is initialized.
- The system must be configured so that allocating memory is not possible after an application is initialized.
- The system must be configured so that system objects and resources cannot be deleted or freed.
- Application tasks must be designed to run forever.
- Because VxWorks 653 might not detect an invalid pointer that an application passes to it, when an application requests that data be stored, it must first check that memory pointers are not corrupted. (The core OS validates all pointers that a partition OS passes to it.)
- Applications must not modify a task control block (TCB) directly, but must use the provided API only.
- Applications must use semaphore types and options that protect against priority inversion.
- Applications must use exclusion mechanisms that protect against deadlock and race conditions.
- All interrupt vectors must have handlers assigned to them.

- Handlers (interrupt, watchdog, and exception) must not call blocking routines.
- The target hardware must have enough CPU power to handle interrupts and to process the computing load.

2

Developing vThreads Applications

2.1	Introduction	13
2.2	vThreads Time Management	15
2.3	Handling External Stimuli	18
2.4	vThreads Memory Management	24
2.5	vThreads Initialization and Restart	25
2.6	Stack Overflow Protection	30
2.7	vThreads Device I/O	33
2.8	vThreads APIs	33
2.9	vThreads System Calls	34

2.1 Introduction

The vThreads partition OS (vThreads) is a multithreading technology that is based on VxWorks 5.5.

This documentation discusses programming concepts for developing applications that run in vThreads partitions.



NOTE: This documentation describes differences between vThreads and VxWorks 5.5, the basics of which are described in [A. VxWorks 5.5](#).

vThreads consists of a kernel plus a subset of the libraries supported in VxWorks 5.5. It has its own priority-preemptive scheduler and its own set of libraries that provide the API. vThreads runs at user level in an application domain under the core OS. One instance of vThreads is completely distinct from both the core OS and other vThreads instances running in other partitions in the same VxWorks 653 module.

Threads

In its partition, vThreads has its own set of objects, including threads. These threads are scheduled by the scheduler associated with each partition. The core OS is unaware of the existence of vThreads threads and the scheduling that happens inside partitions. vThreads threads communicate with the outside world and with other vThreads domains by making system calls to the core OS. For more information, see [2.2.2 vThreads Scheduling](#), p.16.

Memory

On startup, the core OS provides each vThreads partition with a memory heap. vThreads uses this memory heap to manage all allocations required for its objects. This is the only memory vThreads can access. Any memory access outside this range is trapped by the core OS and is illegal. For more information, see [2.4 vThreads Memory Management](#), p.24.

I/O

Unless the I/O device is mapped to the partition, vThreads cannot directly access the device or supervisor-level processor resources. (For more information, see [vThreads Model of Device Drivers](#), p.224.) All I/O, interdomain communication, and so on, are accomplished by system calls to the core OS. A set of system calls is provided for this purpose. For more information, see [2.8 vThreads APIs](#), p.33.

Interrupts and Exceptions

The occurrence of hardware interrupts and exceptions is transparent to vThreads. External events such as clock ticks and status of I/O operations are communicated by the core OS to vThreads by a pseudo-interrupt. The signal handler operates like a hardware interrupt handler: it advances time, manages the delay queue, and unblocks threads waiting for I/O to complete. For more information, see [2.3.1 vThreads Pseudo-Interrupt Signals](#), p.19.

vThreads can be configured to let I/O system calls run asynchronously. If you configure asynchronous I/O, blocking I/O system calls made by vThreads threads are deferred to core OS worker tasks. This lets the partition itself continue running and schedule other vThreads threads while the worker task blocks on the I/O operation. When the I/O completes, the worker task sends a pseudo-interrupt to notify the partition, which in turn unblocks the thread that made the blocking call. For more information on asynchronous I/O, see [9.2 I/O and vThreads](#), p.221.

Pseudo-interrupt and asynchronous I/O facilities give vThreads applications the same view and semantics as if they were running in a VxWorks 5.5 system that controls the hardware directly. On the other hand, the core OS partition scheduler and memory protection facilities ensure the time and space partitioning required of a partitioned operating system are satisfied.

Certification

A subset of vThreads is available that is certifiable to Level A of the RTCA/DO-178B avionics software guidelines.

2.2 vThreads Time Management

vThreads keeps track of the passage of time, which the core OS announces to vThreads by a pseudo-interrupt mechanism similar to a software signal. This is the only major difference in time management between VxWorks 5.5 and vThreads.

Time management within a vThreads partition is accomplished with a single timer queue. This queue manages watchdog timers and timeouts on various operations. It also performs round-robin scheduling of equal priority vThreads threads (if round-robin scheduling is enabled).

2.2.1 vThreads Timer Queue

Elements on the queue are advanced when a system clock tick is announced to vThreads. Each tick denotes the passage of a single unit of time. Ticks are announced to vThreads from the core OS through the pseudo-interrupt mechanism. For information about the pseudo-interrupt mechanism, see [2.3.1 vThreads Pseudo-Interrupt Signals](#), p.19.

There are no limits on the clock tick rate that can be accommodated by vThreads, other than the available processor cycles that can be utilized by the VxWorks 653 module in servicing clock hardware interrupts and issuing pseudo-interrupt signals.

As a performance optimization, multiple clock ticks are announced to vThreads within a single pseudo-interrupt. Rather than announcing every tick into the partition, a single event is issued against the partition only after a specified number of ticks have expired.

The batch delivery of clock ticks lets the core OS conserve processor cycles. Although the core OS is still required to service the clock hardware interrupts, processor cycles are conserved by elimination of the overhead involved in issuing pseudo-interrupt signals, and the subsequent processing of the ticks within vThreads. This is particularly true for systems that require a timeout specification granularity of 0.25 milliseconds (which translates into 4000 ticks per second).

Clock ticks are delivered to a partition only during its window of execution. When the core OS schedules in a new partition, the clock ticks are delivered to the newly scheduled partition.

The issuance of clock ticks to the scheduled-out partition recommences at the start of the partition's next window. At this point, the core OS announces, in batch mode, all the clock ticks that have transpired since the last tick announced to the partition in its previous window. This means that a timeout (or delay) can expire outside the partition's window, but the timeout is acted on only at the beginning of the next partition window.



NOTE: The system integrator decides the system clock rate. It is set in the BSP by calling **sysClkRateSet()**. vThreads can get the clock tick rate by calling **sysClkRateGet()**, but cannot alter it.

2.2.2 vThreads Scheduling

vThreads is comprised of the core portions of VxWorks 5.5, modified to operate solely in a non-privileged processor mode of the CPU (that is, user level). The vThreads scheduler has the same characteristics as the VxWorks 5.5 scheduling algorithm: a priority-preemptive scheduler that allocates the CPU to the highest-priority thread that is ready to run.

Priority-Preemptive Scheduling

vThreads threads are scheduled using a priority-preemptive algorithm by default. The vThreads scheduler uses the priority assigned to each vThreads thread to allocate the CPU to the highest-priority vThreads thread that is ready to run.

Preemption occurs when a thread of higher priority than the running thread becomes ready to run. A higher-priority thread becomes ready to run as a result of either the expiration of a timeout, or the new availability of a resource the thread had been pending on. The preemptive events are delivered from the core OS to the partition, through the pseudo-interrupt mechanism described in [2.3.1 vThreads Pseudo-Interrupt Signals](#), p.19. These events include, but are not limited to, the system clock tick and the system-call completed signals.

The scheduling of equal-priority threads complies with the **SCHED_FIFO** method of the POSIX 13.2.1 specification when round-robin scheduling is disabled. Round-robin scheduling is disabled by default.

In a queue of equal-priority threads, the head of the queue is occupied by the first thread placed on the queue. Threads are placed on the queue in FIFO order. The thread at the head of queue is assigned the CPU when there are no higher-priority threads ready to run.

When the thread is preempted by a higher-priority thread, it remains at the head of the queue of equal-priority threads. Therefore, when the higher-priority thread relinquishes the CPU, that same thread is reassigned the CPU.

When a thread becomes unblocked, it is placed at the tail of the queue of equal-priority threads.

Finally, when the priority of a thread is changed, the thread is also placed at the tail of the queue of equal-priority threads.

This scheduling algorithm is the default one used in VxWorks 5.5. For more information on that algorithm, see [A.2.3 Wind Task Scheduling](#), p.273.

When no threads are ready to run, and no events are left to be processed, the partition is in an idle state. An idle partition does not spin in a tight loop. Rather, it enters the core OS via a system call and blocks there until the core OS sends it one or more events to process. Only then does the partition run again. Events are processed, and any threads that may have become ready to run are run.

Round-Robin Scheduling

The vThreads scheduler provides an optional round-robin scheduling mode. Round-robin scheduling lets the processor be shared fairly by all threads of the same priority. Without round-robin scheduling, when multiple threads of equal priority must share the processor, a single non-blocking thread can usurp the processor until preempted by a thread of higher priority, thus never giving the other equal-priority threads a chance to run.

Round-robin scheduling is disabled by default. It can be enabled or disabled by calling **kernelTimeSlice()**, which takes an argument for the time slice (or interval) that each thread is allowed to run before relinquishing the processor to another equal-priority thread (if the value is zero, round-robin scheduling is turned off).

If round-robin scheduling is enabled and preemption is enabled for the running thread, the system tick handler increments the thread's time-slice count.

When the specified time-slice interval is completed, the system tick handler clears the counter, and the thread is placed at the tail of the list of threads for its priority.

New threads joining a given priority group are placed at the tail of the group, with a run-time counter initialized to zero.

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

If a thread blocks or is preempted by a higher-priority thread during its interval, its time-slice count is saved, and then restored when the thread is eligible to run. In the case of preemption, the thread resumes running once the higher-priority thread completes, assuming that no other thread of a higher priority is ready to run. If the thread blocks, it is placed at the tail of the list of threads for its priority. If preemption is disabled during round-robin scheduling, the time-slice count of the running thread is not incremented.

Time-slice counts are accrued against the thread that is running when a system tick occurs, regardless of whether the thread has run for the entire tick interval. Under certain circumstances, a thread may run for more or less than its allotted CPU time (for example, when preempted by higher-priority threads, or when pseudo-interrupt routines steal CPU time from the thread.)

2.3 Handling External Stimuli

vThreads provides threading capability. It does not provide a mechanism to deliver interrupts from hardware devices requesting service. However, notification of various hardware interrupts and other significant conditions must still be communicated asynchronously from the core OS to the partition. These notifications primarily take the form of pseudo-interrupts. In other words, notifications are delivered asynchronously by the core OS, instead of a hardware interrupt or synchronous exception.

2.3.1 vThreads Pseudo-Interrupt Signals

The occurrence and handling of hardware interrupts is transparent to a partition. Software interrupts are the mechanism by which the core OS notifies a partition of relevant events. Because signals are delivered asynchronously and often reflect the occurrence of hardware conditions, but are not directly raised by a hardware device, they are called pseudo-interrupt signals or pseudo-interrupts.

For a list of pseudo-interrupt event types, see [Table 2-1](#).

A pseudo-interrupt is sent to a partition when the partition needs to process events and attempt thread rescheduling. Sending a pseudo-interrupt always involves queuing an event in the event queue. If necessary, exactly one signal is sent to the partition. No data is sent with this signal. It simply serves to change the control flow, thereby assuring that vThreads reads the event queue and takes the necessary action. If vThreads is in kernel mode, at pseudo-interrupt level, or in the core OS, the control flow naturally processes events and attempts rescheduling, so an additional signal is not required. The pseudo-interrupt handler dequeues the queued event and processes it.

Applications are not allowed to install their own pseudo-interrupt handler. Thus, **intConnect()** is not supported in vThreads. User handlers can be installed only for some event types, using **vThreadsEventHandlerRegister()**. Most events received by the partition are meant to be processed only by the vThreads OS. An attempt to register a user-defined handler for those events returns an error.

Table 2-1 **Pseudo-Interrupt Event Types**

User Handler Forbidden	User Handler Permitted
VT_EVENT_CLOCK_TICK	VT_EVENT_PORT_INT_RECV
VT_EVENT_PARTITION_SAFE_TEXT_IO	VT_EVENT_PORT_INT_SEND

Table 2-1 **Pseudo-Interrupt Event Types** (cont'd)

User Handler Forbidden	User Handler Permitted
VT_EVENT_SC_COMPLETE	VT_EVENT_RELEASE_POINT
VT_EVENT_SYNC	VT_EVENT_USER
VT_EVENT_TIME_MONITOR	
VT_EVENT_WARM_RESTART	

Applications within a vThreads environment communicate with other partitions or the outside world through I/O or through the ports facility. They never directly field an asynchronous signal or event from outside the partition.

The **intLock()** and **intUnlock()** routines are provided to lock out pseudo-interrupts when entering a critical section of code. Typically, a vThreads application calls **intLock()** and **intUnlock()** to prevent interactions with an application routine that is run by a vThreads watchdog timer (**wdLib**).

For performance reasons, the pseudo-interrupt lockout mechanism does not require a system call for each lock-unlock pair. However, if the core OS attempts to deliver a pseudo-interrupt while vThreads is in a critical section, a system call is required to dequeue the event type and associated data when leaving the critical section.

The handling of a pseudo-interrupt often causes a vThreads thread other than the interrupted one to become the highest-priority ready-to-run thread. In such cases, vThreads provides the core OS with an alternative user-level register set to load the vThreads core OS task after the pseudo-interrupt handler has completed. If no alternative register is provided, the core OS uses the register that was saved before delivering the pseudo-interrupt to vThreads.

Various core OS-based tools are provided to interpret vThreads data structures. The core OS issues a pseudo-interrupt prior to accessing any vThreads data structures. This serves to ensure that vThreads is not in a critical section when the core OS tools access the data structures.

If vThreads is running within a critical section, the handling of the pseudo-interrupt is deferred until vThreads exits the critical section. Otherwise, the pseudo-interrupt is handled immediately. Handling of this pseudo-interrupt involves performing a specified core OS system call, thereby indicating that vThreads is not running in a critical section.

The entire vThreads partition blocks during the system call until the core OS has completed accessing the vThreads data structures.

Pseudo-Interrupt Events Forbidden in User Handlers

VT_EVENT_CLOCK_TICK (System Clock Ticks)

The delivery of a clock tick is used to announce the passage of time to vThreads. Each tick denotes the passage of a single unit of time. Multiple ticks may be announced to vThreads within a single pseudo-interrupt.

vThreads maintains a timer queue that is used for managing watchdog timers and timeouts on various operations, and for performing round-robin scheduling of equal-priority vThreads threads (if enabled). Announcing ticks to vThreads is necessary to advance elements in the timer queue as time passes.

For more information, see [2.2 vThreads Time Management](#), p.15.

VT_EVENT_SC_COMPLETE (System Call Complete)

A system call is performed by vThreads to request a service from the core OS. For system calls that block, the core OS assigns a core OS task to complete the request, and returns control to vThreads. vThreads moves the requesting thread from the ready queue to a pend queue, and then schedules the highest-priority thread that is ready to run.

When the assigned core OS task completes the system call, a system-call-complete pseudo-interrupt is issued to inform vThreads of the completion. At this point, vThreads moves the requesting thread back to the ready queue.

For more information, see [2.9 vThreads System Calls](#), p.34.

VT_EVENT_SYNC

This event is used to synchronize vThreads and the core OS when the debugger is active. With this event, vThreads can be stopped in a safe state in which no kernel data is being updated. As a result, the debugger and tools can access vThreads data without the risk of using invalid data structures.

VT_EVENT_WARM_RESTART

When a partition has locked preemption, a warm restart needs to be handled in a cooperative way between the core OS and the partition. This event is sent to the partition when a warm restart is requested and lets data be changed atomically in the partition.

VT_EVENT_TIME_MONITOR

This event is used to notify a partition that time-monitoring information is available.

VT_EVENT_PARTITION_SAFE_TEXT_IO

This event is used to notify a partition that data from application multiplexed I/O is available.

Pseudo-interrupt Events Permitted in User Handlers

VT_EVENT_RELEASE_POINT

This event is used by the vThreads scheduler to start periodic processes. Each window in the schedule is associated with a flag that indicates whether the beginning of the window defines the start of the partition's period. When the flag is set, this event is sent, and the vThreads scheduler starts periodic processes.

VT_EVENT_PORT_INT_RECV (APEX Port RECV)

This event is sent to the partition of a source port after a successful receive operation on one of the destination ports. It permits the partition to asynchronously resume processes that may be blocked on the full source port.

VT_EVENT_PORT_INT_SEND (APEX Port SEND)

This event is sent to the partition of a destination port after a successful send operation. It permits the destination port to asynchronously resume processes that may be blocked on the empty destination port.

VT_EVENT_USER

This event is reserved for application use and is the only event meant to be used with **vThreadsEventHandlerRegister()**.

It is also possible to send a user event from the core OS to a partition by calling **valPseudoInt()**, which can be called only from a kernel protection domain task or an ISR. For details, see the reference entry.

2.3.2 vThreads Synchronous Exception Handling

Synchronous exceptions refer to the class of exceptions that are caused directly by running (or attempted running) an instruction. The synchronous exceptions of interest to vThreads are the ones caused by a programming fault, for example, divide by zero, floating-point exception, data access, or illegal instruction. Synchronous exceptions caused by an MMU TLB miss are not significant for vThreads because they are handled by the core OS.

In addition, handling of vThreads application-generated exceptions in the core OS can be performed only at a fairly coarse level. For example, if a vThreads thread generates a data-access exception, the core OS must suspend the entire partition, and, therefore, all vThreads threads in the partition are suspended. Due to this limitation, exceptions are reported to vThreads so that vThreads can provide a finer-grained handling of the fault.

When an exception is processed, vThreads may choose to reschedule to another thread, resume or restart the thread receiving the exception, restart the partition, or idle, depending on the nature of the exception.

Providing exception notification to vThreads enables vThreads thread-level handling of the fault. The default behavior is to suspend the running vThreads thread, unless the fault occurred during a critical section of the vThreads kernel. In that case the entire partition is restarted. That is, an internally initiated restart is performed.

The same logic that is used in VxWorks 5.5 to process exceptions is used in vThreads, except that the exception stack frame is provided by the core OS rather than by vThreads. As mentioned above, suspending the offending vThreads thread is the default behavior. In addition, a summary of the exception is displayed on the system console. The displayed information includes the offending vThreads thread ID, the exception type, the program counter of the vThreads thread when the fault occurred, and additional information depending on the exception type.

In addition to the exception diagnosis performed by VxWorks 5.5, vThreads also diagnoses vThreads thread stack-overflow conditions. If enabled, core OS support provides a system call to enable and disable memory protection for guard pages at the end of each vThreads thread stack.

As with VxWorks 5.5, an application can register a handler to be run whenever any vThreads thread in the partition generates a fault. This lets an application perform its own fault handling and bypass the default vThreads behavior.

2.4 vThreads Memory Management

The vThreads memory manager is the same as VxWorks 5.5, in other words, **memPartLib.c**. Applications in a partition call **malloc()** or **memPartAlloc()** to allocate memory dynamically from the partition system heap (which is the only heap).

In addition, vThreads lets applications disable dynamic allocations when a partition has completed booting and reached its normal operating mode. The **memPartAllocDisable()** routine serves this purpose. In addition, it is up to the application to allow the disabling of dynamic allocations after bootup and initialization. Disabling dynamic allocation can be done for each partition at configuration and build time by setting the **allocDisable** parameter to **true** in the XML configuration file. For more information, see the *VxWorks 653 Configuration and Build Reference*. To get the value at run-time, call **configRecordFieldGet()** with the **PARTITION_ALLOC_DISABLE** selector.

If the partition configuration selects this feature, and once **memPartAllocDisable()** has been called in the partition, any subsequent call to partition-level allocation or free routines results in an error. In addition, **errno** is set to **S_memLib_FUNC_NOT_AVAILABLE**. If health monitoring is configured into the VxWorks 653 module, a health monitor event is logged. Once disabled, allocations cannot be enabled except by a partition restart.

After dynamic allocation has been disabled, the following **memLib** routines return **NULL**, with **errno** set as described above:

- **calloc()**
- **cfree()**
- **free()**
- **malloc()**
- **memalign()**
- **memPartAlignedAlloc()**
- **memPartAlloc()**
- **memPartFree()**
- **memPartRealloc()**
- **realloc()**
- **valloc()**

APEX applications typically issue the **SET_PARTITION_MODE** service to alter the partition operating mode. When set to **NORMAL** mode, **memPartAllocDisable()** is called automatically to prevent additional memory allocations.

C applications can call **memPartAllocDisable()** directly when they have allocated all the memory they expect to need. The routine takes no arguments and returns nothing.

As a consequence of disabling dynamic allocation, no vThreads objects, such as threads, semaphores, or message queues, can be created. Hence, partitions start booting with allocations enabled and call **memPartAllocDisable()** only when they are ready to do so.

Disabling allocation for each partition is independent of other partitions and the core OS. The core OS disables allocation when it has booted and created all partitions and their associated infrastructure. A partition disables allocation after its own initialization is complete.

2.5 vThreads Initialization and Restart

The core OS creates and launches each partition as a native core OS task. The core OS is not aware of the threads and scheduling that occur inside partitions after they are started. The entry point for the partition is always the boot code for vThreads. vThreads runs in user mode as a core OS task. After vThreads initialization is complete, vThreads starts additional partition-level components and user applications.

2.5.1 vThreads Boot Sequence

Once started, each partition initializes itself as follows:

1. The first step in the vThreads boot sequence involves making a system call to the core OS to find required initialization parameters. The core OS transfers the following initialization parameters as part of the **SYSINFO_GET** system call:
 - a. partition heap start address and size
 - b. number of worker tasks if any
 - c. partition operating mode (that is, **PARTITION_IDLE**, **PARTITION_COLD_START**, or **PARTITION_WARM_START**)
 - d. reason or type of the last restart

- e. maximum number of core OS files the partition can open
- f. system clock rate
- g. MMU page size
- h. memory allocation disable flag
- i. copy of the partition configuration record in the partition space
- j. list of shared data regions used by the partition
- k. partition symbol table
- l. list of initialization routines for the partition's shared library and application components

Most parameters are specific to a partition and can vary from one partition to another. Others, including the system clock rate and MMU page size, are global to the VxWorks 653 module and do not vary.

2. The vThreads kernel initializes itself in the given heap of memory and enters multi-tasking mode by starting **tRootTask**.
3. **tRootTask** carries on system initialization by initializing other OS facilities such as intertask communication mechanisms, the I/O system, exception handling, and signal handling. vThreads proper is now functional and ready to initialize additional OS layers.
4. The POSIX libraries are initialized (if selected) or the APEX facilities (if selected). Initialization of APEX facilities involves further system calls to get the partition configuration record information for APEX object initialization.
5. Finally, vThreads initializes any application components configured into the VxWorks 653 module.
6. The application code is now running.

This vThreads boot sequence is a modified version of the VxWorks 5.5 boot sequence. All hardware and processor initialization steps have been handled in one of the following ways:

- Removed entirely, because they are no longer relevant in the context of a partitioned operating system.
- Removed because the core OS performed them when it booted.
- Replaced by a system call to the core OS.

Additionally, a few vThreads-specific steps and ordering have been added as applicable.

Individual vThreads threads are not schedulable entities for the core OS. All vThreads thread scheduling is done entirely by vThreads in user mode. The vThreads scheduler is the same as the VxWorks 5.5 scheduler, which is a priority-preemptive scheduler with an option to enable round-robin scheduling for all threads at a given priority level. All resource allocations performed by vThreads are made from its assigned memory pool. This pool (the partition heap) is defined by the system integrator and remains fixed at that size. In other words, the partition heap cannot dynamically grow beyond the size set in the configuration file.

2.5.2 vThreads Restart

A partition restart can be initiated either externally (for example, by the core OS) or by the partition itself. A partition can choose to restart itself in response to exceptional conditions, application errors, and so on. Restarts can be of two types: cold or warm. In both cases, vThreads is rebooted, but some subsequent actions are not performed for warm restarts.

Cold Versus Warm Restarts

When a restart is externally initiated (either cold or warm), any outstanding system calls in progress are aborted before the partition restarts itself. All restart activity is performed within the partition's schedule window. No dynamic memory allocation is performed during a partition restart.

A cold partition restart assumes that the partition has corrupted itself irretrievably. The following actions are performed:

- Outstanding system calls are flushed.
- vThreads task is stopped.
- Partition memory is cleared.
- Text and data sections for the partition executable are reloaded.
- The uninitialized data area (**.bss**) is cleared.
- Per-client data for any attached shared libraries is reloaded.
- Persistent data sections are reloaded.
- vThreads boots again from the original entry point.
- Devices controlled by the partition (if any) are re-initialized.

A warm partition restart assumes that only the partition's application has been corrupted. The following actions are performed:

- Outstanding system calls are flushed.
- vThreads thread is stopped.
- Non-persistent data sections, including those from any attached shared libraries, are reloaded.
- The uninitialized data area (**.bss**) is cleared.
- vThreads boots again from the original entry point.

Applications can choose to bypass certain steps in their initialization sequence on warm restarts. In other words, the application developer is the one who specifies the initialization sequences for both cold and warm starts, and can design both options to best serve the application.

Cooperative Warm Partition Restart Mechanism

When a warm partition restart is requested, the vThreads abstraction layer (VAL) **valPartitionRestart()** routine performs the VAL- and vThreads-related portion of a restart operation. This routine issues a **VT_EVENT_WARM_RESTART** pseudo-interrupt event into the partition.

The core OS task that is performing the restart operation (the partition restart task) delays until an acknowledgement is received from the partition. If the delay expires without an acknowledgment, a cold partition restart is performed.

For a non-APEX partition (POSIX or vThreads partition), the **VT_EVENT_WARM_RESTART** pseudo-interrupt handler in vThreads uses the preemption lock count of the running thread to determine whether the partition is in a critical section. Normally, **taskIdCurrent** indicates the running task, except in the case where a process-level health monitor thread has forcefully preempted a preemption-locked thread. In this case, **taskIdCurrent** indicates the thread that the health monitor thread preempted.

For an APEX partition, the **VT_EVENT_WARM_RESTART** pseudo-interrupt handler uses the lock level of the APEX partition to determine whether the partition is in a critical section.

If the partition is in critical section, a global variable in the partition is set to indicate that a warm partition restart has been requested. This variable is always checked by **taskUnlock()**. If a warm partition restart has been requested, **taskUnlock()** performs a **SYSCALL_WARM_RESTART_ACK** system call to unblock

the core OS partition restart task as soon as the blocking call is complete. At this point, the core OS proceeds with the warm restart operation. If the partition is not in a critical section, the pseudo-interrupt handler immediately performs the **SYSCALL_WARM_RESTART_ACK** system call to unblock the core OS partition restart task.

There is no need for the **VT_EVENT_WARM_RESTART** pseudo-interrupt handler to check whether the thread has locked out pseudo-interrupts. When the application eventually unlocks pseudo-interrupts, via **intUnlock()**, vThreads services any pending events in the queue. When the **VT_EVENT_WARM_RESTART** event is encountered, a **SYSCALL_WARM_RESTART_ACK** system call is performed (assuming that the application did not also lock preemption).

This mechanism lets applications trust their critical data across warm partition restarts. The purpose of a warm restart, as opposed to a cold restart, is to preserve certain sections of memory (ELF sections with a specific name, such as **.persistent.data**) over the restart. An application could not trust the integrity of critical data over warm restarts if a warm restart could occur while a thread is in the midst of updating that data. A cooperative mechanism lets that data be trusted.

A cooperative warm partition restart does *not* occur if the partition is running a **SYSCALL** method even though an application thread may have previously locked preemption. Thus, applications should not call routines that may result in a **SYSCALL** method call after locking preemption.

The partition's **watchDogDuration** parameter in the XML configuration file (see the *VxWorks 653 Configuration and Build Reference*) determines the length of partition time that the core OS waits for an acknowledgement from the **VT_EVENT_WARM_RESTART** event. To get the value at run-time, call **configRecordFieldGet()** with the **PARTITION_WD_DURATION** selector. If vThreads fails to acknowledge in the specified time frame, warm partition restart is escalated to a cold partition restart.

Comparison with Non-Cooperative Warm Partition Restart

For situations where a partition itself requests a warm restart, through the **SYSCALL_PARTITION_MODE_SET** system call, the core OS partition restart task does not block during the invocation of **valPartitionRestart()**. Also, if the partition requests a partition restart, for example, due to a fatal error, while the core OS partition restart task is waiting for an acknowledgement, **valPartitionRestart()** unblocks so that the warm restart can proceed.

Partition Restart and Device Drivers

When a driver resides completely in the core OS, issues arise between device use by partitions and a partition restart. To ensure reliability in terminating system calls in progress, core OS device drivers that are used by partitions should follow the following rules to make the VxWorks 653 module safe during partition restart:

- The driver's **open()**, **creat()**, **remove()**, and **close()** routines should be deterministic in execution and bounded in time. They should not block for an arbitrarily long time (in other words, they should not have unbounded execution characteristics).
- The **FIORESET** ioctl command code should be supported by the device driver. It is called during restart of partition, if the driver was in the midst of a **read()**, **write()**, or **ioctl()** operation on the device.
- The **FIORESET** command should make the thread of control that is running the driver's **read()**, **write()**, or **ioctl()** operation complete. This could involve either:
 - Wake up the partition thread if it was blocking on I/O and make it return from the I/O operation.
 - If the thread is in the middle of the I/O operation but is not blocking, it could do a **longjump()** from the thread so that it returns from the I/O operation.

FIORESET must never terminate a thread that is performing an I/O operation.



NOTE: Partition-based I/O does not have these constraints because it is reset during restart.

2.6 Stack Overflow Protection

Stack overflow protection is a feature new to vThreads, one not present in VxWorks 5.5. It consists of one or more pages of memory at the top of the stack that cannot be written to. If a write operation is attempted, a stack overrun error is issued.

2.6.1 Guard Pages

The number of guard pages can be set for each partition at configuration and build time by setting the **numStackGuardPages** parameter in the XML configuration file. For more information, see the *VxWorks 653 Configuration and Build Reference*. To get the value at run-time, call **configRecordFieldGet()** with the **PARTITION_NUM_STK_GUARD_PAGES** selector.

The **taskSpawn()** routine rounds the thread stack size up to the nearest page boundary. The amount of memory consumed by the guard pages is added automatically to the requested thread stack size. You need not factor the guard page size into your stack size calculations.

The stack guard page is accessible from supervisor level only. Any access to this page from a vThreads thread (for example, by a stack overflow) causes a data access exception.

Guard pages are created and their access permissions set when a vThreads thread is created.

When a thread that is running in user mode (in other words, not running a system call) overflows its stack, a data access exception is generated by the CPU. As with all other synchronous exceptions (except debugging-related exceptions), the exception information is passed up to vThreads. The exception is delivered to the offending thread where a **SIGBUS** signal is delivered if a signal handler has been registered (**sigaction**) for this signal. If a signal handler has not been registered for the offending thread, a HM event injection occurs with the **HM_STACK_OVERFLOW** code.

User-level read and write privileges are removed from the guard pages; however, supervisor read and write privileges remain. This prevents an access exception due to stack overflow during a system call. When a system call is issued, the VAL and core OS operate on the stack of the calling thread.

If the VAL or core OS generate an access exception due to a stack overflow, the entire module is rebooted. In fact, a cold system restart is performed. In contrast, if a thread overflows its stack, vThreads or the application can issue a partition restart in the worst case.

2.6.2 Defaults

The default number of stack guard pages is one for a PowerPC and zero for the simulator. Stack guard pages are disabled for the simulator because the MMU page

size for a Pentium is 64 KB. Due to this large size, too much memory is consumed by guard pages, which thus easily causes memory exhaustion within the partition.

The number of stack guard pages is defined by the `NUM_STACK_GUARD_PAGES` parameter. For information on setting parameters, see the *VxWorks 653 Configuration and Build Guide*. For information on a particular parameter, see the *VxWorks 653 Configuration and Build Reference*.

2.6.3 Limitations

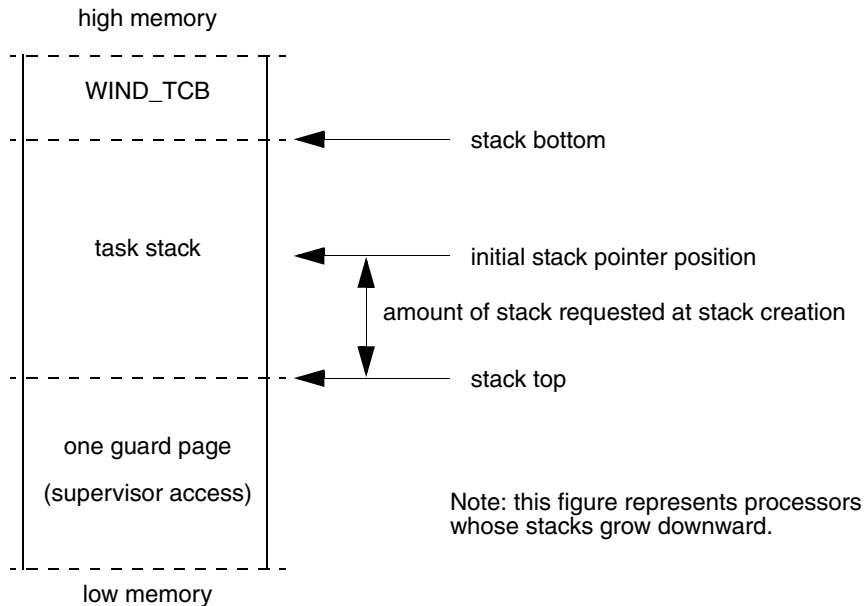
The following limitations of the guard-page method of stack overflow protection should be kept in mind:

- If a system call overflows the guard page, the entire VxWorks 653 module is rebooted using a cold restart (in the case of the simulator only).
- Underflow conditions cannot be caught.
- Overflows of huge stack frames that may cross over the guard page completely cannot be caught.
- There is no user-callable API. Guard pages are always set.
- The initial thread spawn sets as many bytes above the start of the guard page as the stack size requested for the thread. Therefore, asking for a small amount, such as 100 bytes, gives you one page of memory, but any access above 100 bytes causes a trap. Calculate your sizes carefully.
- A stack overflow exception is treated by vThreads as a synchronous exception. These exceptions are handled by vThreads in the same way as other synchronous exceptions.
- The guard pages are disabled when the exception handler is running in the partition for stack overflow exceptions. Therefore, stack overflow detection is disabled while the stack overflow handler runs.



CAUTION: If `setjmp()` and `longjmp()` are used to recover from a stack overflow exception, you must be sure that there are at least 1170 bytes of stack memory remaining between the `setjmp()` call and the end of the stack. This is because `longjmp()` uses the context saved by `setjmp()` to re-enable stack overflow detection.

Figure 2-1 **Stack Guard Pages in Memory**



2.7 vThreads Device I/O

For details of vThreads device I/O, see [9.2 I/O and vThreads](#), p.221.

2.8 vThreads APIs

vThreads provides threading facilities as well as facilities for intertask and interpartition communication.

vThreads provides APIs for the following:

- vThreads (similar to VxWorks 5.5)

- ANSI
- utility libraries to support buffer management, linked lists, ring buffers, and the event logging (for the Wind River System Viewer)
- debugging library (**userlib**) for the target shell
- show routines
- POSIX (if **INCLUDE_POSIX** is included in the VxWorks 653 module)

For detailed information about the libraries and their routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

2.9 vThreads System Calls

Examples of system services provided by vThreads include **open()**, **read()**, **write()**, **close()**, and **ioctl()**, I/O system routines. These are used by the vThreads device driver to access non-local devices.

A partition's access to system calls to the core OS is granted by permission bitmasks, which are defined by the partition's **syscallPermissions** parameter in the XML configuration file. For more information, see the *VxWorks 653 Configuration and Build Reference*. The **SYSCALL_ALL_PERMISSION** bitmask grants access to all methods. For more information, see [System Call Permission Bitmasks](#), p.138. To get the value at run-time, call **configRecordFieldGet()** with the **PARTITION_SC_PERMISSION** selector.

3

Developing COIL Applications (Core OS Interface Library)

- 3.1 Introduction 35
- 3.2 VxWorks 653 Architecture and COIL 36
- 3.3 Accessing Core OS Services 37
- 3.4 Communicating with Other Partitions 37
- 3.5 Handling Interrupts and Exceptions 38
- 3.6 Restarting COIL Partitions 41
- 3.7 Device I/O in COIL Partitions 41
- 3.8 Monitoring Health in COIL Partitions 41
- 3.9 COIL API 41

3.1 Introduction

The core OS interface library (COIL) is a partition OS that provides the minimum necessary functionality to let an application communicate with the core OS. The library routines are independent of the vThreads partition OS.

This documentation discusses programming concepts for writing applications that run in COIL or COIL-based partitions.

COIL supports the following:

- interpartition messaging via ARINC ports
- management of interrupts and exceptions
- device I/O
- injection of health monitoring events

COIL is often augmented, and the result is called the user partition OS. For example, if an APEX service for ports is required, the user partition OS must provide it. It is also the responsibility of the user partition OS to provide any additional management required around the provided COIL API. An application in a COIL partition can use either straight-line code or implement its own process-scheduling mechanism.

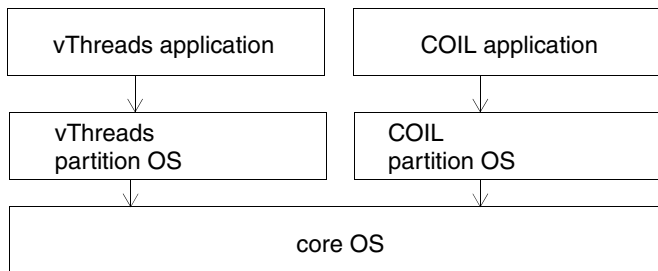
COIL and the Core OS Certifiable Subset

If the application calls a COIL library routine that in turn calls a routine no longer in the core OS due to certification, the COIL routine fails gracefully.

3.2 VxWorks 653 Architecture and COIL

Figure 3-1 shows the relationship between applications, partition OSs, and the core OS.

Figure 3-1 Relationship between Applications, Partition OSs, and the Core OS



Linking with Shared Libraries

A COIL partition is configured the same way as a vThreads partition, except the linkage path for the COIL partition is set to the COIL partition OS instead of to a vThreads partition OS.

COIL supports shared libraries, except in the case where the shared library requires code from vThreads. Shared libraries attached to a COIL partition cannot have any dependency on a vThreads partition OS. That is, any core OS accesses from a COIL-attached shared library must use the COIL API and not the vThreads API. Access restrictions and configuration information for COIL-dependent shared libraries are identical to vThreads shared libraries. As well, COIL partitions can access shared data regions in the same manner as vThreads partitions, with all the same access restrictions and configuration information.

For information on linking to shared libraries, see the *Workbench User's Guide, VxWorks 653 Version*.

For information on programming in vThreads partitions, see [2. Developing vThreads Applications](#).

3.3 Accessing Core OS Services

A COIL application's ultimate accesses to core OS services is controlled using the same mechanism as for vThreads. Specifically, the system call permissions specified when the system is configured indicate which core OS calls can be made on behalf of the application. For information of configuring system call permissions, see the *VxWorks 653 Configuration and Build Reference*.

3.4 Communicating with Other Partitions

COIL supports interpartition messaging through ARINC 653 ports. As a result, COIL partitions can communicate with each other. In addition, they can communicate with APEX partitions (and vThreads partitions that have minimal APEX support) and vice versa. However, the API to access ports does not comply

with the ARINC 653 specification; that is, it is not APEX. If this compliance is needed, the user partition OS must provide the appropriate APEX API. For information on the APEX API for ports, see [4.7 Communicating between Partitions](#), p.67.

Ports for COIL partitions are configured in the same manner as ports for APEX partitions. For details, see the *VxWorks 653 Configuration and Build Guide*.

Ports in COIL partitions are subject to the same access restrictions as ports in APEX partitions. For information, see [4.7 Communicating between Partitions](#), p.67.

Calls for interpartition messaging are non-blocking. However, pseudo-interrupts may be used to avoid constant polling, as described in the following sections.

Avoiding Polling When Destination Buffers Are Full

During a send operation to a port, the destination buffer might be full. If this is the case, the send operation returns a failure (the operation is non-blocking). When a message is subsequently removed from the destination buffer, the `COIL_EVENT_PORT_INT_RECV` pseudo-interrupt is sent to the source partition. At this point, the application can call `coilPortIntRecv()` to determine which port the pseudo-interrupt applies to, and it can then resume sending messages.

Avoiding Polling When Incoming Message Queues Are Empty

During a read operation from a port, the incoming queue might be empty. If this is the case, the read operation returns a failure (the operation is non-blocking). When the source partition subsequently adds a message to the incoming queue, the `COIL_EVENT_PORT_INT_SEND` pseudo-interrupt is sent to the destination partition. At this point, the application can call `coilPortIntSend()` to determine which port the pseudo-interrupt applies to, and it can then resume reading messages.

3.5 Handling Interrupts and Exceptions

[Table 3-1](#) lists the routines that provide pseudo-interrupt and exception handling in a COIL partition. For details, see the `coilLib` reference entry in the *VxWorks 653 vThreads API Reference*.

As for a vThreads partition OS, a COIL partition OS includes a copy of run-time data for each attached partition. Therefore, each partition can install, or not install, its own pseudo-interrupt handler, exception handler, or both.

Table 3-1 **COIL Interrupt and Exception Handling Routines**

Routine	Description
coilExcConnect()	Defines a routine to handle exceptions in the partition (optional). For information on the behavior if an exception handler is not defined, see 3.5.2 Handling Exceptions , p.40.
coilIntConnect()	Defines a routine to handle pseudo-interrupts in the partition (optional). For information on the behavior if a pseudo-interrupt handler is not defined, see 3.5.1 Handling Pseudo-Interrupts , p.39.
coilIntLock()	Prevents all pseudo-interrupts from occurring in the partition.
coilIntTickGet()	Returns the number of ticks received so far by the partition.
coilIntUnlock()	Lets pseudo-interrupts occur in the partition.

3.5.1 Handling Pseudo-Interrupts

COIL always handles some pseudo-interrupts. If the user partition OS has defined its own pseudo-interrupt handler (by calling **coilIntConnect()**), COIL forwards the remaining pseudo-interrupts to this handler. If the handler is not defined, COIL discards the events. [Table 3-2](#) lists the only events that user-defined handlers need to handle.

Table 3-2 **Events that User-Defined Pseudo-Interrupt Handlers Need to Handle**

Pseudo-Interrupt Event	Meaning
COIL_EVENT_CLOCK_TICK	Timer ticks have been received.
COIL_EVENT_PORT_INT_RECV	Messages have been removed from a full port message queue.

Table 3-2 **Events that User-Defined Pseudo-Interrupt Handlers Need to Handle** (cont'd)

Pseudo-Interrupt Event	Meaning
COIL_EVENT_PORT_INT_SEND	Messages have been sent to an empty port message queue.
COIL_EVENT_RELEASE_POINT	The scheduler major frame has started.
COIL_EVENT_SC_COMPLETE	A pending system call has completed.
COIL_EVENT_USER	User-defined pseudo-interrupt event that can be used by the user partition OS.
COIL_EVENT_WARM_RESTART	Warm restart has been requested. The application should call coilWarmRestartAck() when it is ready to be restarted.

When the handler returns, the context that it preempted is restored.

Pseudo-interrupts can be locked by calling **coilIntLock()**. When pseudo-interrupts are locked for a partition, the partition is not preempted to deliver incoming pseudo-interrupts. Instead, pseudo-interrupts destined for the partition are queued until they are subsequently unlocked. When they are unlocked, **coilIntUnlock()** calls the pseudo-interrupt handler to process queued pseudo-interrupt events.

As for vThreads partitions, the queued pseudo-interrupts for a COIL partition cannot be flushed. They must be processed by the partition.

3.5.2 Handling Exceptions

The user partition OS can optionally define its own exception handler by calling **coilExcConnect()**. If a handler is so defined, COIL calls it when an exception occurs and passes the exception information to the handler. If the handler returns and the exception is not fatal, the context that it preempted is restored.

If the user partition OS does not define an exception handler, all exceptions are considered fatal. That is, when an exception is detected from a partition, the core OS suspends the partition.

3.6 Restarting COIL Partitions

A COIL partition can be restarted by calling `coilPartitionModeSet()` and specifying an operating mode as is done for APEX partitions. For information on restarting APEX partitions, see [4.4.5 Setting the Partition Mode](#), p.49.

For warm restarts, the COIL pseudo-interrupt event, `COIL_EVENT_WARM_RESTART`, is passed to the partition. This pseudo-interrupt event gives the partition some time to perform any cleanup activities that might be necessary before the partition is restarted. The partition is expected to perform the necessary cleanup and then respond by calling `coilWarmRestartAck()`.

3.7 Device I/O in COIL Partitions

For information, see [9.4 I/O and COIL](#), p.264.

3.8 Monitoring Health in COIL Partitions

For information, see [8.7 Health Monitoring for COIL Partitions](#), p.217.

3.9 COIL API

For details about the COIL API, see the reference entries for `coilLib` in the *VxWorks 653 vThreads API Reference*.

4

Developing APEX Applications

(ARINC 653 API)

- 4.1 Introduction 43
- 4.2 Adding APEX Support to vThreads Partitions 45
- 4.3 Terminology and Concepts: APEX Versus vThreads 45
- 4.4 Managing APEX Partitions 46
- 4.5 Managing APEX Processes 51
- 4.6 Managing Time in APEX Partitions 61
- 4.7 Communicating between Partitions 67
- 4.8 Communicating with Other Modules 74
- 4.9 Communicating within APEX Partitions 77
- 4.10 Monitoring Health in APEX Partitions 86

4.1 Introduction

APEX is an API between an application program and an operating system that supports the ARINC 653 specification. For VxWorks 653, the operating system is the vThreads partition OS, and ultimately the core OS. The major enhancement APEX brings to a vThreads partition is in time and process management and the ability to manage periodic and aperiodic processes and their associated deadlines.

This chapter discusses programming concepts for writing APEX applications that run in vThreads partitions in a VxWorks 653 module. It explains the Wind River implementation of APEX. It does not discuss what is included in the ARINC 653 specification. If you need that level of detail, read the specification before you read this chapter.

With the addition of an APEX component, an application in a vThreads partition can use the full or partial (called minimal) APEX API. This documentation calls a vThreads partition with *full* APEX support an APEX partition.

In addition to APEX interfaces, an APEX partition has access to the vThreads API. However, except where noted, this chapter describes APEX interfaces only. For information on the vThreads API, see [2.8 vThreads APIs](#), p.33.

APEX Services

Full APEX support provides services to do the following:

- Manage partitions.
- Manage processes.
- Manage time.
- Communicate with other partitions (using messages, ports, and channels).
- Communicate within partitions (using buffers, blackboards, semaphores, and events).
- Monitor health.

Some of these services (such as communicating within and outside partitions) can instead be handled using vThreads or POSIX objects (such as pipes, message queues, and semaphores). However, such an implementation does not comply with the ARINC 653 specification.

Minimal APEX support provides services to do the following:

- Manage partitions.
- Communicate between partitions (using messages, ports, and channels).

4.2 Adding APEX Support to vThreads Partitions

A VxWorks 653 module (or part of one) can have either full or minimal APEX support, but not both. In this documentation, only a partition with full APEX support is called an APEX partition.

To provide a vThreads partition with the full set of APEX services, include the **INCLUDE_APEX** component in one or more of the following domains:

- vThreads partition
- one or more shared libraries with which the vThreads partition links
- the partition OS with which the vThreads partition links

The domain cannot include **INCLUDE_POSIX**. The resulting VxWorks 653 module cannot include **INCLUDE_APEX_MINIMAL**.

To provide a minimal APEX interface, include the **INCLUDE_APEX_MINIMAL** component in one or more of the above domains. The domain can include **INCLUDE_POSIX**. In other words, a vThreads partition or POSIX partition can have minimal APEX support. The resulting VxWorks 653 module cannot include **INCLUDE_APEX**.

4.3 Terminology and Concepts: APEX Versus vThreads

Some of the terminology in this chapter is specific to the ARINC 653 specification and APEX. [Table 4-1](#) lists some ARINC 653 terms and concepts and their vThreads equivalents.

Table 4-1 Terminology and Concepts: APEX Versus vThreads

Term or Concept	APEX	vThreads
Service or routine	Service	Routine
	ALLCAPS	mixedCase()
	ACTION_OBJECT (for example CREATE_PROCESS)	objectAction() (for example taskDelete())

Table 4-1 **Terminology and Concepts: APEX Versus vThreads** (cont'd)

Term or Concept	APEX	vThreads
	Services are issued or requested	Routines are called
Schedulable unit	Process	Task (APEX processes are implemented as vThreads tasks)
Scheduling method	FIFO	Priority-preemptive (FIFO) or round-robin
Priority numbering	Higher the value, higher the priority	Lower the value, higher the priority (0 is the highest priority)
Buffer	Buffer	N/A
Event	Event	vThreads event

4.4 Managing APEX Partitions

Managing a partition includes allocating partition memory and initializing the partition in accordance with the ARINC 653 specification.

4.4.1 Allocating Partition Memory

Each partition has predetermined areas of physical memory allocated to it. These unique memory spaces vary in size based on the requirements of the individual partitions. At most, one partition has write access to any particular area of memory. Memory partitioning is ensured by prohibiting write access outside a partition's defined memory areas. To ensure complete separation of applications, read access is also prohibited outside a partition.

4.4.2 Initializing Partitions: Cold and Warm Starts

Whereas the resource allocation necessary for each partition is specified in the XML-based configuration and build process (see the *VxWorks 653 Configuration and Build Guide*), the corresponding objects are defined when the partition is initialized. The core OS exclusively controls the allocation of resources to the partition by reserving specific memory. The partition uses this reserved memory to create the specified objects.

COLD_START

The cold-start partition operating mode is used when a partition is created and when the VxWorks 653 module starts from a powered-off state. During a cold start, partition objects are allocated and initialized.

WARM_START

The warm-start partition operating mode causes a partition to be re-initialized or restarted because of an error. During a warm start, persistent data is not re-initialized, and the partition code is not reloaded.

4.4.3 Partition Attributes

Partition attributes are defined in the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Reference*.

Fixed Partition Attributes

- **Identifier**

Uniquely defined within a VxWorks 653 module, and used to facilitate activating the partition and routing messages.

- **Memory requirements**

The amount of physical memory to be allocated and mapped for the partition.

- **Period**

The activation period of the partition. It is used to determine the partition's run-time placement within the core OS overall time frame.

- **Duration**

The amount of processor time the core OS gives to the partition every period of the partition.



NOTE: The partition scheduler in the core OS does not use period and duration attributes directly. It only ensures that periods and durations are compatible with specified schedules. Up to 16 schedules can be defined, which allows for time partitioning where a partition can be activated several times within a major time frame. Scheduler validation can be disabled by a system parameter.

- **Criticality level**
The RTCA/DO-178B criticality level of the partition (from A down to E).
- **Communication requirements**
Those partitions with which the partition communicates by linking it to communication channels.
- **Partition health monitor table (health monitor configurations)**
Instructions to the health monitor on the actions required. For example, the health monitor supervisory facility can restart the partition in response to a fatal fault.

4.4.4 **Getting Partition Status**

The `GET_PARTITION_STATUS` service gets the partition status of the current condition (also called the start condition or the partition mode reason). [Table 4-2](#) lists the possible values and their meanings.

The `apexPartitionModeReasonPtrGet()` routine gets a pointer to the partition mode reason. This routine is not an APEX service.

Table 4-2 **APEX Partition Status Values**

Partition Status (Start Condition, Partition Mode Reason)	Reason for Current Partition Mode
HM_MODULE_RESTART	Recovery action taken at the VxWorks 653 module level.
HM_PARTITION_RESTART	Recovery action taken at the partition level.
NORMAL_START	Power-up.

Table 4-2 APEX Partition Status Values (cont'd)

Partition Status (Start Condition, Partition Mode Reason)	Reason for Current Partition Mode
PARTITION_RESTART	Request for a COLD_START or WARM_START partition mode.
POWER_ERROR_RESTART	(A Wind River extension to the ARINC 653 specification.)

4

4.4.5 Setting the Partition Mode

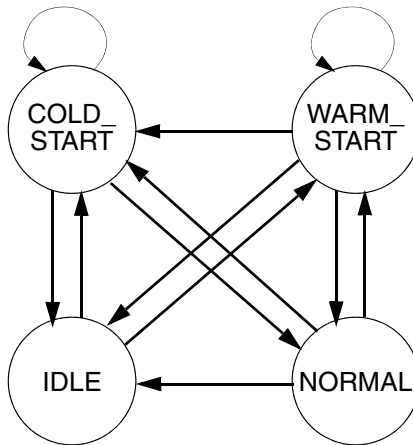
The SET_PARTITION_MODE service sets the operation mode for the current partition. Table 4-3 lists the available modes and their resulting actions.

Table 4-3 APEX Partition Modes

Partition Mode	Resulting Action
COLD_START	The partition restarts using the cold start initialization sequence.
IDLE	The partition shuts down. The partition is not initialized (that is, none of the ports associated with the partition are initialized), no processes are running, but the time windows allocated to the partition are unchanged.
NORMAL	The activate process is scheduled.
WARM_START	The partition restarts using the warm start sequence.

Figure 4-1 shows the allowable transitions in a partition’s operating mode.

Figure 4-1 APEX Partition Mode Transitions



NOTE:
It is not possible to go from
COLD_START to WARM_START
directly or through a transition to IDLE.

4.4.6 Controlling Preemption in Partitions

A process can issue the **LOCK_PREEMPTION** service (defined in the **apexProcess** library) to lock preemption in the partition. The service increments the lock level of the partition and disables processes from being rescheduled in the partition. This ability is important when processes are accessing critical sections or resources that are shared by multiple processes in the same partition. These critical sections may be specific areas of memory, certain physical devices, or the normal calculations and activity of a particular process.

The ability to intervene with normal rescheduling operations does not imply that the application is directly controlling vThreads. Since vThreads provides this service and knows all resulting actions and effects beforehand, the integrity of vThreads is not affected.

In addition, the **LOCK_PREEMPTION** service does not affect the scheduling of other partitions: if a process within a critical section is interrupted when the partition window ends, that process is the first to run when the partition runs again.

A process can issue the **UNLOCK_PREEMPTION** service (defined in the **apexProcess** library) to unlock preemption in the partition. The service decrements the lock level of the partition. Rescheduling of processes resumes only when the lock level is zero.

The `partitionCurrentLockLevelPtrGet()` routine gets a pointer to the partition's current lock level. This routine is not an APEX service.



NOTE: Preemption locking does not prevent the error handler process from running.

4.4.7 **Setting New Partition Schedules**

The `SET_SCHEDULE_MODE` service selects a new schedule (including an empty schedule) and transition time for a partition. [Table 4-4](#) lists the available scheduling modes.

 Table 4-4 **APEX Scheduling Modes**

Scheduling Mode	The Transition Is Effective at the:
TRANSITION_MAJOR	Major frame boundary
TRANSITION_MINOR	End of the current window
TRANSITION_TICK	Next clock tick



NOTE: The `SET_SCHEDULE_MODE` service is not part of the ARINC 653, Part 1 specification, but is being considered for a subsequent edition.

4.5 **Managing APEX Processes**

APEX processes are programming units contained within an APEX partition. Each process runs concurrently with other processes in the same partition. A process consists of the following:

- the executable program
- data and stack areas
- program counter

- stack pointer
- priority deadline

4.5.1 Creating Processes

The **CREATE_PROCESS** service is used to create a process with certain attributes and allocate resources for it. Since the service can be called only during warm or cold start of a partition, creation attributes cannot be changed after a partition is initialized. Each process is created only once during the life of the partition. Also, all the processes in a partition must be defined in such a way that the necessary memory resources for each process can be determined at system build time. For information of configuring memory, see the *VxWorks 653 Configuration and Build Guide*.

For information on getting creation attributes dynamically, see [4.5.4 Getting the Current Status of Processes](#), p.53.

The following names are field names in a **PROCESS_ATTRIBUTE_TYPE** structure, which is an argument to the **CREATE_PROCESS** service.

BASE_PRIORITY

Process initial priority.

DEADLINE

Type of deadline (**SOFT**, **HARD**, or no deadline. This indicates the correct remedial action to the health monitor.

ENTRY_POINT

Starting address of the process.

NAME

String identifier for the process. It must be unique within the partition.

PERIOD

Delay between two activations (for periodic processes only).

STACK_SIZE

Size (in bytes) of the stack allocated to the process.

TIME_CAPACITY

The elapsed time within which the process should complete running.

4.5.2 Changing the Current Priority of Processes

Although the initial priority is set when the process is created (**BASE_PRIORITY**), the current priority can be changed dynamically through the **SET_PRIORITY** service.

For information on getting the current priority, see [4.5.4 Getting the Current Status of Processes](#), p.53.

4.5.3 Increasing Deadline Times

Deadline time is the absolute time by which the process should be complete. It starts as the return value of the **GET_TIME** service (current system time) plus the **TIME_CAPACITY** that is specified when the process is created. vThreads periodically evaluates deadline time to determine whether the process is satisfactorily completing its processing within the allotted time (time capacity). Deadline time can be increased by issuing the **REPLENISH** service (defined in the **apexTime** library).

For information on getting the value of deadline time, see [4.5.4 Getting the Current Status of Processes](#), p.53.

4.5.4 Getting the Current Status of Processes

The **GET_PROCESS_STATUS** service gets the current status of a process. The return value is of **PROCESS_STATUS_TYPE** type, which contains the fields listed in [Table 4-5](#).

Table 4-5 **APEX Process Status Information**

Field in PROCESS_STATUS_TYPE	Description
ATTRIBUTES	The creation attributes for the process. See 4.5.1 Creating Processes , p.52.
CURRENT_PRIORITY	Current priority of the process. See 4.5.2 Changing the Current Priority of Processes , p.53.

Table 4-5 **APEX Process Status Information** (cont'd)

Field in <code>PROCESS_STATUS_TYPE</code>	Description
<code>DEADLINE_TIME</code>	Current deadline time for the process. See 4.5.3 Increasing Deadline Times , p.53.
<code>PROCESS_STATE</code>	Current state of the process. See 4.5.9 Process State Transitions , p.56.

4.5.5 Getting Process IDs

The `GET_MY_ID` service gets the process ID of the calling process.

The `GET_PROCESS_ID` service gets the process ID of the process with the specified name.

4.5.6 Getting and Using vThreads Task Information

Because APEX processes are implemented as vThreads tasks, they have vThreads task IDs. The following routines are available:

The `taskIdFromProcIdGet()` routine gets the vThreads task ID for the specified APEX process ID.

The `procIdFromTaskIdGet()` routine gets the APEX process ID for the specified vThreads task ID.

4.5.7 Types of Processes

There are two types of processes:

- **Periodic processes**

A periodic process is a process that is activated at regular times (defined by the **PERIOD** creation attribute). At activation time, the process becomes eligible for scheduling. When that happens, the state of the process changes to **RUNNING**, or **READY** if it is preempted by a higher-priority process (periodic or not).

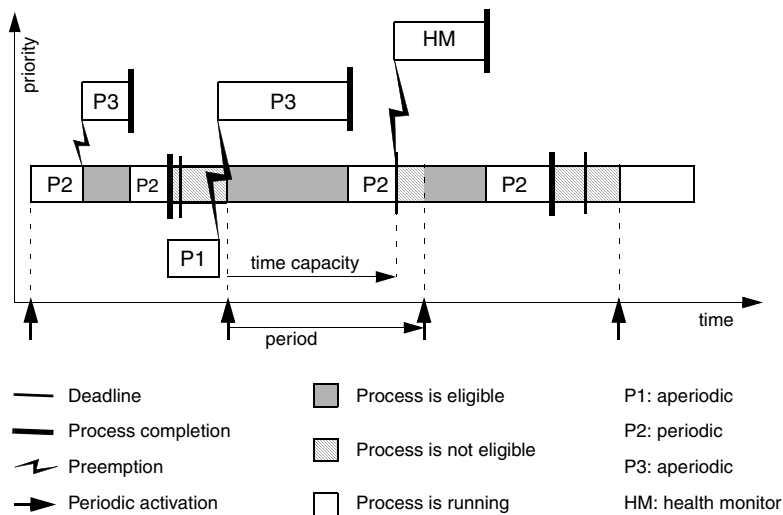
▪ Aperiodic processes

An aperiodic process is the same as a periodic process, but without an activation time. That is, an aperiodic process has a **PERIOD** creation attribute equal to **INFINITE_TIME_VALUE**.

4.5.8 Scheduling Processes

Figure 4-2 illustrates an example of process scheduling. Processes are scheduled according to the POSIX **SCHED_FIFO** method. In the example, P2 does not complete during the second time period. It is preempted by P3 most of the time, so it misses its deadline.

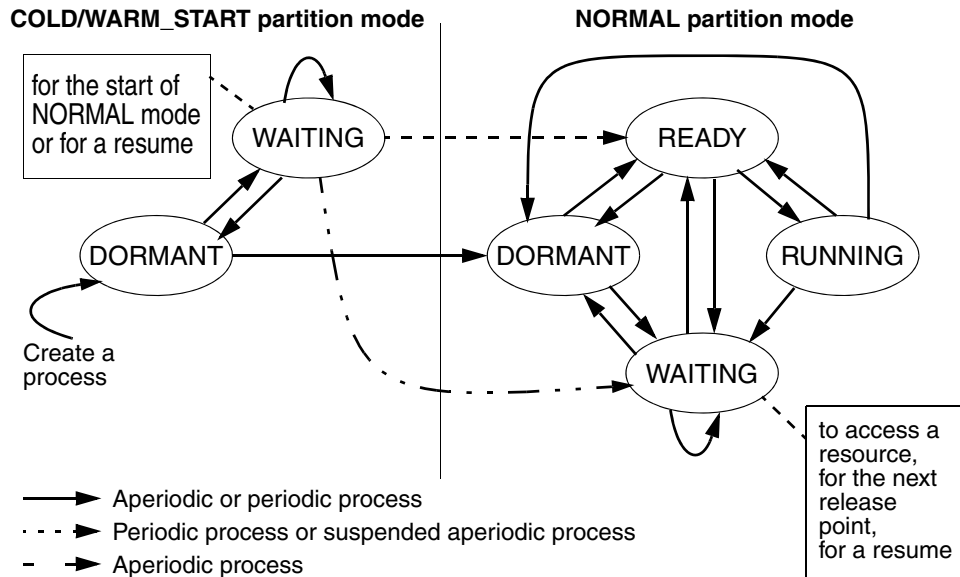
Figure 4-2 Example of Processes Scheduled with FIFO Scheduling



4.5.9 Process State Transitions

Figure 4-3 shows the relationships between the various states for APEX processes.

Figure 4-3 APEX Process State Transitions



DORMANT State

The **DORMANT** state indicates that the process is ineligible to receive resources. A process is in the **DORMANT** state before it is started and after it is terminated (that is, after a **STOP** service has been issued). Processes are created in the **DORMANT** state.

The **DORMANT** state moves to the following states:

READY

- When a process is started by another process while in **NORMAL** mode.

WAITING

- When a process is started during **INIT** mode.
- When a periodic process is started during **NORMAL** mode and is waiting for its next release point.

- When an aperiodic process is started with a delay during **NORMAL** mode.

DORMANT

- When a process is not started before the transition to **NORMAL** mode.

WAITING State

The **WAITING** state indicates that the process is not allowed to receive resources until a particular event occurs. The process is waiting for one or both of the following reasons:

- It is waiting on a resource, such as a semaphore or an event.
- It is suspended.

The **WAITING** state moves to the following states:

DORMANT

- When another process stops the process.
- When an error occurs and the health monitor stops the process.

READY

- When an unavailable resource becomes available and the process is not suspended.
- When the **RESUME** service is requested for the process and the process is not waiting for any resources.
- When the **TIMED_WAIT** service was requested, and the delay has expired.
- For a periodic process, when the time to activate the process (deadline time) is reached.
- For an aperiodic process started during **INIT** mode (which is started in the **WAITING** state), when the partition enters **NORMAL** mode.

WAITING

- When a process that is waiting to access a resource (delay, semaphore, period, event, message, and so on) is suspended.
- When a process that is both waiting to access a resource and suspended is resumed, or the resource becomes available, or the timeout expires.

- For a periodic process started during **INIT** mode (which is started in the **WAITING** state), when the VxWorks 653 module enters **NORMAL** mode.
- For an aperiodic process started using the **DELAYED_START** service in the **INIT** mode, when the partition enters **NORMAL** mode.

RUNNING State

The **RUNNING** state indicates that the process is running. Only one process can be running at a time. The previous state of a running process is always **READY**.

The **RUNNING** state moves to the following states:

DORMANT

- When the running process stops itself.
- When the health monitor stops the process because an error occurred.

READY

- When the running process requests a service delay with a delay time of zero (this is equivalent to round-robin scheduling of processes of the same priority).
- When another process with a higher priority enters the **READY** state.

WAITING

- When the running process suspends itself.
- When the running process attempts to access an unavailable resource (semaphore, buffer, event, blackboard, queuing port) with a non-zero timeout.
- When the running process requests a delay service (such as a timed wait or periodic wait) with a non-zero delay.

READY State

The **READY** state indicates that the process is eligible to be scheduled and is waiting to run. The process is not running for either or both of these reasons:

- A higher-priority process is running.
- Preemption is locked.

The **READY** state moves to the following states:

DORMANT

- When another process issues a **STOP** service on the process.
- When the health monitor stops the process because an error occurred.

RUNNING

- When the scheduler selects the process to run. (This is the only way to enter the **RUNNING** state.)

WAITING

- When another process issues a **SUSPEND** service on the running process.

4.5.10 Suspending and Resuming Processes

When a process is suspended, the process is not allowed to run, and its state is **WAITING** until another process resumes it. When a process waits on a resource such as a semaphore or an event, it can also be suspended. The services to suspend or resume processes are:

- **SUSPEND_SELF**

If the current process is an aperiodic process, the service suspends it until **RESUME** is issued or the specified timeout expires.

If preemption is disabled and the process being suspended is the one holding the preemption lock, **SUSPEND_SELF** returns **INVALID_MODE**.

- **SUSPEND**

The service lets the current process suspend any aperiodic process, except itself, until another process resumes the suspended process. If the process is pending in a queue at the time it is suspended, it stays in the queue. When it is resumed, it continues pending unless it was removed from the queue before the end of its suspension. The process might have been removed if a particular condition occurred, a timeout expired, or the queue was reset.

A process may suspend any other process asynchronously.

If process B suspends an already suspended or self-suspended process A, the service has no effect.

If preemption is disabled and the process being suspended is the one holding the preemption lock, **SUSPEND** returns **INVALID_MODE**.

- **RESUME**

The service lets the current process resume another previously suspended process. The resumed process becomes ready if it is not waiting on a resource such as a delay, semaphore, period, event, or message. Since a periodic process cannot be suspended, it cannot be resumed.

4.5.11 Stopping and Starting Processes

The **STOP** service makes a process ineligible for processor resources (in other words, its state is **DORMANT**) until another process issues the **START** service (causing the first process to enter the **READY** state). After it is created, a process is also in a **DORMANT** state. The **DELAYED_START** service applies only to periodic processes and lets process schedules be phased. The services to stop and start processes are:

- **STOP_SELF**

The service lets the current process stop itself. If the current process is not the error handler process, the partition is placed in the unlocked condition. The service should not be called when the partition is in **WARM_START** or **COLD_START** modes. If it is, the behavior is not defined.

- **STOP**

The service makes a process ineligible for processor resources until another process issues **START**. The **STOP** service lets the current process abort any process except itself. When a process aborts another process that is pending in a queue, the aborted process is removed from the queue.

- **START**

The service initializes all attributes of a process to their default values and resets the process's run-time stack. If the partition is in **NORMAL** mode, the process's deadline expiration time and next release point are calculated. The service lets the current process start another process.

- **DELAYED_START**

The service initializes all attributes of a process to their default values, resets the processes's run-time stack, and places the process in the **WAITING** state (that is, the specified process goes from **DORMANT** to **WAITING**). If the partition is in **NORMAL** mode, the process's release point is calculated with the specified delay time. In addition, the process's deadline expiration time is calculated. The service lets the current process start another process.

4.5.12 Controlling Preemption

Preemption locking or unlocking disables or enables process rescheduling in a partition. For details, see [4.4.6 Controlling Preemption in Partitions](#), p.50.

4.6 Managing Time in APEX Partitions

Time partitioning is a major characteristic of VxWorks 653 and all ARINC 653 systems.

4.6.1 Scheduling Partitions

For information on how the core OS schedules partitions, see [7.13 Partition Scheduling](#), p.169.

4.6.2 System Clock Time

The system clock time gives the unique time of the system. Time is unique and independent of partition execution within a VxWorks 653 module. All time values or capacities are related to this unique time and are not relative to any partition execution. The timer provides the time of day, and it is used as a stamp or for anything that needs time or date information.

The `GET_TIME` service gets the system clock time.

For a description of system time, see [7.12 System Time](#), p.169.

4.6.3 Requesting Resources and Timeouts

When a process requests an APEX resource (such as a semaphore or an event), it can specify a timeout of one of the following types:

INFINITE_TIME_VALUE

Never expire. It is equivalent to wait forever.

ZERO_TIME_VALUE

Do not wait for the resource. If the resource is not available, return an error.

Finite value of timeout

The maximum amount of time to wait for a resource.

The timeout's units are of **SYSTEM_TIME_TYPE** type, which defines time in nanoseconds. Although expressed in nanoseconds, the time is rounded up to ticks, and the actual timeout is a multiple of the system clock period.

A timeout usually results in an early return from the service and a **TIMED_OUT** return code.

If a timeout expires outside the partition window, it is acted on at the beginning of the next partition window (as with deadlines; see [4.6.5 Deadlines](#), p.62).

4.6.4 Scheduling Processes

APEX time management services let partitions control their processes.

At the end of each processing cycle, a periodic process requests the **PERIODIC_WAIT** service to get a new deadline. The new deadline is calculated from the next periodic release point for that process. For all processes, the **TIMED_WAIT** service lets the process suspend itself for a minimum amount of elapsed time. After the wait time has elapsed, the process becomes available to be scheduled.

The **REPLENISH** service lets a process postpone its current deadline by the amount of time that has already passed.

Each process in a partition can specify an amount of elapsed time (called the time capacity) it is allowed to consume in order to satisfy its processing requirement. This time capacity is used to set a processing deadline time that vThreads periodically evaluates to determine whether the process is satisfactorily completing its processing within the allotted time.

4.6.5 Deadlines

Each process has associated with it a fixed time capacity, which represents the response time allotted to it for satisfying its processing requirements.

The deadline time (a variable process attribute) determines whether the process is satisfactorily completing its processing within its time capacity. Deadline time can be increased by issuing the **REPLENISH** service. The **PERIODIC_WAIT** service cancels the current deadline, and a new deadline is created at the next activation.

A deadline can expire outside the partition window, but it is acted on at the beginning of the next partition window (as with timeouts).

There are three types of deadlines:

- **Hard deadlines**

If a process fails to meet a hard deadline within the specified time period, vThreads takes remedial action.

- **Soft deadlines**

If a process fails to meet a soft deadline within the specified time period, typically, the failure is recorded and processing continues.

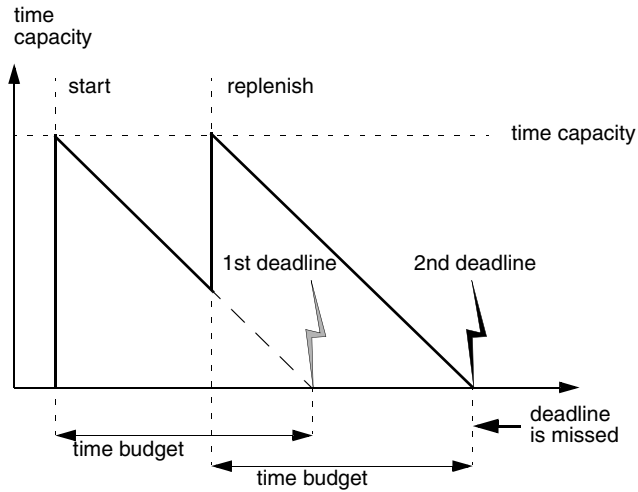
- **No deadline**

No action is taken if a process fails to complete processing within the specified time period.

For a periodic process, the countdown on deadline time starts when the process's active period starts. Countdown is disabled when the process requests the **PERIODIC_WAIT** service. Deadline time is ended when the process is stopped or when it calls the **REPLENISH** or **PERIODIC_WAIT** services. Countdown is deactivated when the partition is in an operating mode other than **NORMAL**.

For an aperiodic process, the deadline time countdown starts when the process starts and the partition mode is **NORMAL**. The deadline time is rearmed with additional time equal to the time budget specified when the process requests a **REPLENISH** service (see [Figure 4-4](#)). The process can specify the additional time. A periodic process deadline cannot be postponed beyond its next release point.

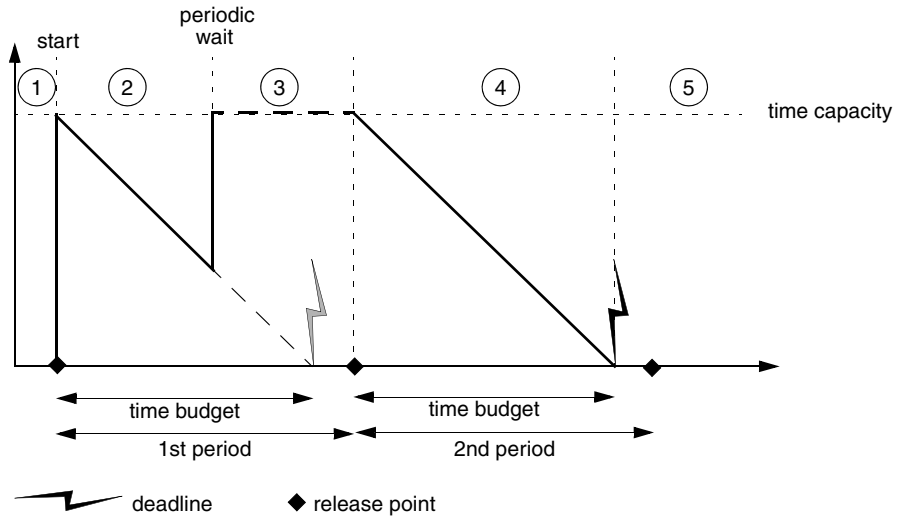
Figure 4-4 **Process with Replenish (Periodic or Aperiodic Processes)**



NOTE: To make [Figure 4-4](#) through [Figure 4-6](#) easier to read, the time capacity (the time initially allotted to the process to complete its work) and the amount of time added by **REPLENISH** (specified by the **BUDGET_TIME** parameter) appear to be equal and constant. In reality, **BUDGET_TIME** can be larger or smaller than the time capacity.

The deadline ends when the process is stopped, or when the partition state is not **NORMAL**.

Figure 4-5 Periodic Process Examples with PERIODIC_WAIT and Deadline



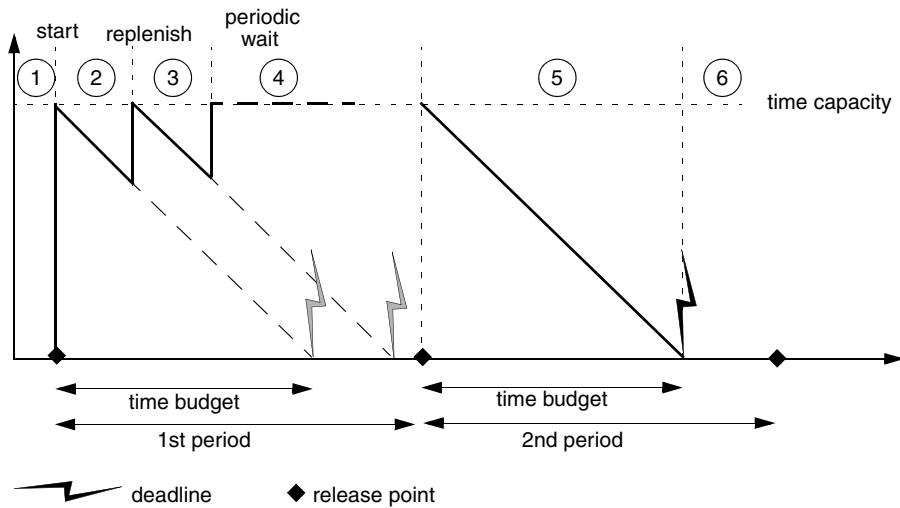
In [Figure 4-5](#), the periodic process is in the following states:

1. **DORMANT.**
2. **RUNNING** (or **READY** if another process has preempted it).
3. **WAITING.**
4. **RUNNING** (or **READY** if another process has preempted it).
5. **READY** until the health monitor takes an action. The process has not completed within the deadline interval.

In [Figure 4-6](#), the periodic process is in the following states:

1. **DORMANT.**

Figure 4-6 Periodic Process with REPLENISH and PERIODIC_WAIT



2. **RUNNING** (or **READY** if another process has preempted it).
3. **RUNNING** (or **READY** if another process has preempted it).
4. **WAITING**.
5. **RUNNING** (or **READY** if another process has preempted it).
6. **READY** until the health monitor takes an action. The process has not completed within the deadline interval.

4.6.6 Release Points

The first release point for a periodic process is relative to the start of its partition's first window in the major time frame. Using a **DELAY** in the **DELAYED_START** service enables phasing of the process schedule. Subsequent release points are based on the previous release point and the process period.

4.7 Communicating between Partitions

Interpartition communication includes all communication between two or more partitions in a VxWorks 653 module.

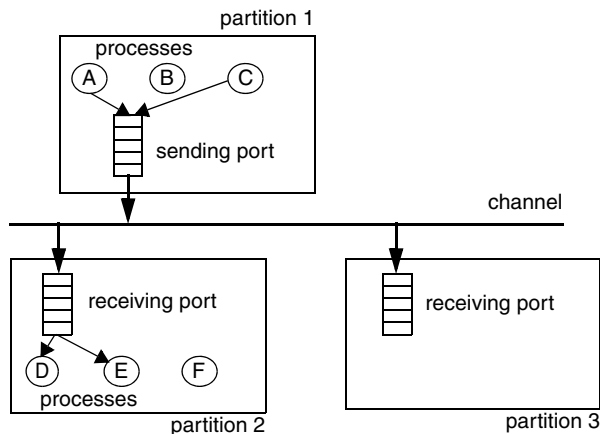
APEX partitions within a VxWorks 653 module communicate with each other by messages, ports, and channels. The same is true for vThreads partitions that have access to the minimal APEX services. In addition, APEX partitions can communicate with COIL partitions. For details, see [3.4 Communicating with Other Partitions](#), p.37.

For information on communicating outside the VxWorks 653 module, see [4.8 Communicating with Other Modules](#), p.74. The API for communicating between partitions and for communicating outside the module is the same. However, under some circumstances, there are minor behavioral differences when communicating outside the module. For details, see [4.8 Communicating with Other Modules](#), p.74.

A message can be sent from one source port to one or more destination ports. Processes read from these destination ports.

The configuration of this messaging system is defined when the VxWorks 653 module is configured. For more information, see the *VxWorks 653 Configuration and Build Guide*.

Figure 4-7 Sending Messages between Partitions



4.7.1 Limitations of APEX for Communicating between Partitions

Although the ARINC 653 standard specifies the following, APEX does not support them:

- multicast and client-server messages
- acknowledgement of messages

4.7.2 APEX Messages

APEX messages are contiguous blocks of data.

Although the ARINC 653 standard lets messages be decomposed into small blocks, communicated individually, and re-assembled before delivery, VxWorks 653 transmits full messages only. This avoids checking for message completeness and retransmitting segments.

Messages can be of fixed or variable lengths. Fixed length means a fixed size for every occurrence of a particular message. A variable-length message can vary in size. The sender specifies the length when the message is sent. To accommodate various message lengths, the messaging system allocates resources for the maximum length, which is defined in the ARINC 653 standard.

A message can be sent periodically or on demand (aperiodically). The messaging system operates independently of the content of the messages it transmits.

Any given message can be sent from a single partition only. In addition, once delivered, messages are destroyed. That is, it is not possible to request old versions of messages.

4.7.3 APEX Channels

A channel defines the following:

- logical link between one source port and one or more destination ports
- mode of transfer of the messages from the source to the destination
- characteristics of the messages to be sent

A message sent to one destination port is called a directed message. A message sent to multiple destination ports is called a broadcast message.

Each channel can be configured to operate in a specific mode. Two modes of transfer are defined: sampling mode and queuing mode. The messaging service returns an error if the port configuration (mode, direction) is not compatible with the request.

For information on how to configure ports and their associated channels in an XML configuration file, see the *VxWorks 653 Configuration and Build Guide*.

The consistency of the configuration is checked at built time and startup time. For a channel, the size of the sending port cannot exceed the size of any of the receiving ports.

Sampling Mode

In sampling mode, messages typically carry similar, but updated, data. No queuing is performed. A message remains in the source port until it is sent or overwritten. Messages arrive at the destination port or ports in the order in which they were sent. Each new message overwrites the previous one when it reaches the destination port and remains there until it is overwritten itself. Sampling mode supports variable-length messages.

Refresh Rate

Refresh rates applies to destination ports in sampling mode. The refresh rate indicates the maximum acceptable age of a valid message, from the time it was received at the port. It is specified when the port is created. When the message is read, a validity output parameter indicates whether the age of the message is consistent with the port's refresh rate.

Queuing Mode

In queuing mode, each new instance of a message may contain uniquely different data. Therefore, overwriting previous messages is not allowed during the transfer. Messages are queued in the source port until they are sent, and no message is lost (except in the case of a full message queue with the **RECEIVER_DISCARD** protocol). Messages are stored in the receiver port until a process reads them. For information about the protocols required to manage message queues, see [Port Protocols](#), p.72. Queuing mode supports variable-length messages.

4.7.4 Ports

A port can be one of the following general types:

- **Local Ports**

Local ports are APEX ports that let applications communicate with each other within a VxWorks 653 module. They are attached to partitions within the module.

- **Pseudo-Ports**

Pseudo-ports are used to communicate outside the VxWorks 653 module. A pseudo-port connects a port to a driver, using a specialized API. For information on pseudo-ports, see [4.8 Communicating with Other Modules](#), p.74.

- **Direct-Access Ports**

Direct-access ports implement APEX queuing ports without software buffering. They are also used to communicate outside the VxWorks 653 module. A channel that has a direct-access port must have a single source and destination. Direct-access ports can be in partitions or pseudo-partitions. For details on those in partitions, see [4.8.2 Communicating Through Direct-Access Ports in a Partition](#), p.76. For details on those in pseudo-partitions, see [4.8.1 Communicating Through Pseudo-Ports in a Pseudo-Partition](#), p.74.

- **Null Ports**

Null ports are APEX ports that are used to ease incremental system integration, where some part of a system may not be present during the integration of other parts. To this end, a channel can initially be configured with a null source port, which is equivalent to a port that is always empty. Also, a channel can initially be configured with one, some, or all null destination ports, which are equivalent to ports that are always ready to accept data and always consume it without error. Null ports can be attached to partitions, the core OS, or pseudo-partitions. For information on pseudo-partitions, see [4.8 Communicating with Other Modules](#), p.74.

The same APEX messaging services can be used for all types of ports.

Ports are defined by a set of unique attributes that are specified by the ARINC 653 standard. The attributes are specified in the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*.

To get a value at run-time from the core OS, call **configRecordFieldGet()** with **PORT_CFG_RECORD** and the appropriate field selector, as shown below:

CFG_PORT_CHANNEL

Parent channel.

CFG_PORT_DIRECTION

SOURCE or **DESTINATION**.

CFG_PORT_DRIVER_NAME

The name of the driver (for pseudo-ports only).

CFG_PORT_MAPPING

Whether the port is one of:

- direct-access pseudo-port in the core OS (**DIRECT_ACCESS_PORT**)
- local port in a partition (**LOCAL_PORT**)
- null port in the core OS or a partition (**NULL_PORT**)
- pseudo-port in the core OS or a partition (**PSEUDO_PORT**)

CFG_PORT_MODE

QUEUEING or **SAMPLING**.

CFG_PORT_MODULE

Parent VxWorks 653 module.

CFG_PORT_MSG_SIZE

Maximum size (in bytes) of a message that can be sent or received by this port.

CFG_PORT_NAME

Port name, 1 to **MAX_NAME_LENGTH** (as set in **apexType.h**)
NULL-terminated ASCII string.

CFG_PORT_NB_MSGS

Message queue size (for queueing ports only).

CFG_PORT_PARTITION

Parent partition, as defined in the partition definition record
(**PARTITION_CFG_RECORD**).

CFG_PORT_PROTOCOL

Port message protocol. One of **RECEIVER_DISCARD**, **SENDER_BLOCK**, or
NOT_APPLICABLE. For more information, see [Port Protocols](#), p.72.

CFG_PORT_REFRESH

Port refresh rate in **SYSTEM_TIME_TYPE** increments (for sampling ports only).

Port Protocols

The ARINC 653 standard does not specify port protocols, but VxWorks 653 does provide the following:

SENDER_BLOCK

A queuing message is sent to all the channel's destination ports. If any one is full, the message is queued in the source port in FIFO order.

When the source port is full and if a timeout was specified, sender processes are blocked during the **SEND_QUEUING_MESSAGE** service.

When a destination port is emptied, retransmission is attempted. Whether it succeeds depends on the state of the channel's other destination ports.

The main advantage of using the **SENDER_BLOCK** protocol is that no messages are lost, as the ARINC 653 specification requires. The main drawback is that it introduces coupling between partitions. A nonresponsive receiving partition blocks the entire channel, affecting the normal behavior of other receiving partitions.

Because the **SENDER_BLOCK** attribute that is set on a single destination port can block the entire channel, it is considered a channel-wide attribute and is attached to the channel's source port.

RECEIVER_DISCARD

If one of the channel's destination ports is full, the source port discards the message for that port. Therefore, if all the destination ports are full, the message might be lost. When a message is so discarded, the port's overflow flag is set to notify the application of the discarded (lost) message.

The **RECEIVER_DISCARD** protocol avoids the problem caused by a faulty application that fails to read or empty its destination ports, thereby preventing other partitions from receiving messages.

4.7.5 Working with Queuing Messages

Creating Queuing Ports

The **CREATE_QUEUING_PORT** service creates an empty port in queuing mode and returns a port ID. The **QUEUING_DISCIPLINE** attribute indicates whether blocked processes are queued in FIFO or priority order.

Sending Queuing Messages

The `SEND_QUEUING_MESSAGE` service sends a message to the specified queuing port. If there is sufficient space at the queuing port, the message is added to the end of the port's queue. If there is insufficient space, the sending process is blocked and added to the sending port's queue, according to the port's queuing discipline. The process stays on the queue until the specified timeout expires (if it is finite) or until space for the message becomes free at the queuing port.

Receiving Queuing Messages

The `RECEIVE_QUEUING_MESSAGE` service receives a message from the specified queuing port. If the queuing port is not empty, the message at the head of the port's queue is removed and returned to the caller. If the queuing port is empty, the process is blocked until the specified timeout or until a message arrives.

Getting Queuing Port Information

The `GET_QUEUING_PORT_ID` service gets the port ID for a specified queuing port name.

The `GET_QUEUING_PORT_STATUS` service gets the following information for a queuing port:

- direction
- number of messages at the port
- number of waiting processes
- size

4.7.6 Working with Sampling Messages

Creating Sampling Queues

The `CREATE_SAMPLING_PORT` service creates an empty sampling port and returns a port ID.

Writing Sampling Messages

The `WRITE_SAMPLING_MESSAGE` service writes a message at the specified sampling port, overwriting a previous message.

Reading Sampling Messages

The `READ_SAMPLING_MESSAGE` service reads a message at the specified sampling port and returns a validity parameter that indicates whether the age of the message is consistent with the port's refresh rate. The age is the difference between the value of the system clock when the message is written into the port and the value of the system clock when the messages is read at the destination port.

Getting Sampling Port Information

The `GET_SAMPLING_PORT_ID` service gets the port ID for a specified sampling port name.

The `GET_SAMPLING_PORT_STATUS` service gets the following for a sampling port:

- direction
- refresh rate
- size
- validity of the last message read by the specified sampling port

4.8 Communicating with Other Modules

Applications can use the APEX message services to communicate with other modules. The services are the same services that applications use to communicate with other partitions within the module. For information on the API as it relates to communicating within a module, see [4.7 *Communicating between Partitions*](#), p.67.

4.8.1 Communicating Through Pseudo-Ports in a Pseudo-Partition

For an application to use APEX queuing-message services to communicate with other modules through pseudo-ports in pseudo-partitions, the platform provider needs to do the following:

- **Configure the remote port**

Configure the remote port in the pseudo-partition of the partition's communication channel as a pseudo-port or a direct-access port. For

configuration details, see the port and channel documentation in the *VxWorks 653 Configuration and Build Guide*. An application cannot determine which type of ports it is communicating through. If the remote port is a direct-access port, the application may see differences compared to using a pseudo-port that is not direct access. For details, see [Communicating Through Direct-Access Ports in a Pseudo-Partition](#), p.75.

- **Supply a driver**

Supply a supervisor-level port driver in the core OS. For information on writing such a driver and adding it to the core OS, see [7.15 Setting up Communication with Other Modules](#), p.185.

Communicating Through Direct-Access Ports in a Pseudo-Partition

If the remote port of an application's channel is a direct-access port in a pseudo-partition, the application can send and receive queuing messages at any time in its partition window.

If an application specifies a non-zero timeout for sending or receiving messages, the timeout is ignored and treated as zero.

Receiving Messages

[Table 4-6](#) shows what happens when an application issues the `RECEIVE_QUEUING_MESSAGE` service on a channel with a direct-access port attached to a pseudo-partition.

Table 4-6 **Receiving Messages on a Channel with a Direct-Access Port in a Pseudo-Partition**

Message?	Time Enough in Partition Window to Read?	Result of Issuing <code>RECEIVE_QUEUING_MESSAGE</code>
No	N/A	Service immediately returns the <code>NOT_AVAILABLE</code> APEX return-code parameter to the application. (The return code for a pseudo-port that is not direct access would be <code>TIMED_OUT</code> .)

Table 4-6 **Receiving Messages on a Channel with a Direct-Access Port in a Pseudo-Partition** (cont'd)

Message?	Time Enough in Partition Window to Read?	Result of Issuing <code>RECEIVE_QUEUING_MESSAGE</code>
Yes	Yes	Service immediately returns the message to the application.
Yes	No	vThreads retries the operation until the end of the partition window, rereading the message at the start of the partition's next window.

Sending Messages

If an application issues the `SEND_QUEUING_MESSAGE` service on a channel with a direct-access port in a pseudo-partition and there is not time for the core OS to complete the write operation, vThreads retries the operation until the end of the partition window, rewriting the message at the start of the partition's next window.

4.8.2 Communicating Through Direct-Access Ports in a Partition

For an application to use APEX message services (for queuing, sampling, and SAP messages) to communicate with other modules through direct-access ports in its partition, the following must be done:

- **Configure ports**
The platform provider and system integrator configure the channel's ports in the partition. For details, see the port and channel documentation in the *VxWorks 653 Configuration and Build Guide*.
- **Supply a driver**
The platform provider supplies a user-level port driver in the partition. The interface is the same as the interface for the supervisor-level port driver in the core OS. For information on writing a supervisor-level driver for the core OS, see [7.15 Setting up Communication with Other Modules](#), p.185. Consider the following differences as you follow this information:
 - Because the port driver is in a partition, you can use only the vThreads API.
 - vThreads passes to the driver the time remaining in the partition window. This value is always infinite.

Sending and Receiving Messages

An application cannot determine which type of port it is communicating through. The only difference between the behavior of message services that use APEX ports and those that use direct-access ports occurs when an application issues a service with a non-zero timeout. In this case, the service returns an **INVALID_PARAM** return-code parameter.

4.9 Communicating within APEX Partitions

APEX provides the following APEX objects so that processes can communicate with each other within a partition:

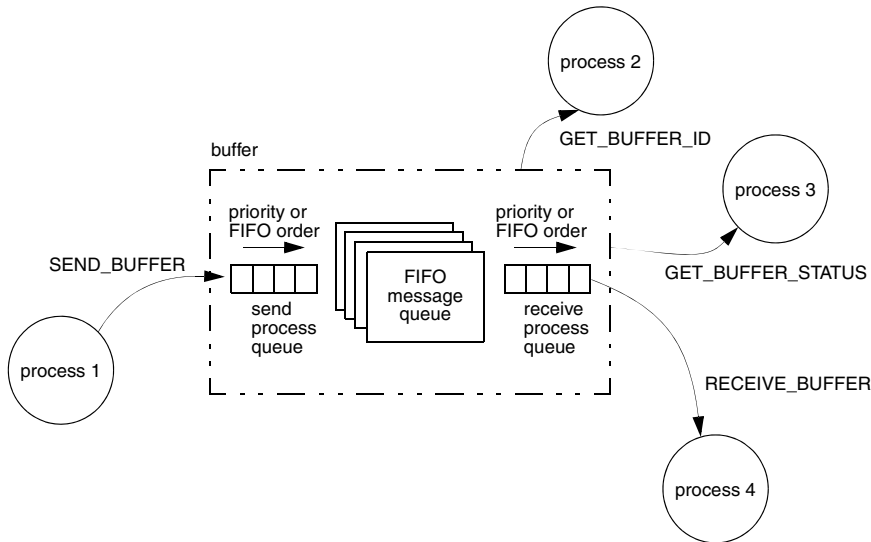
- buffers
- blackboards
- semaphores
- events

4.9.1 Communicating Using APEX Buffers

APEX buffers let processes communicate with each other within a partition. Buffers support a single message type between multiple source and destination processes. Communication is indirect: participating processes address the buffer rather than the opposing processes directly, thus providing a level of process independence.

Buffers store multiple messages in message queues and no messages are lost. [Figure 4-8](#) summarizes how processes use a buffer to communicate.

Figure 4-8 Processes Using a Buffer to Communicate



Creating APEX Buffers

APEX buffers can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many buffers as are supported by the memory that is pre-allocated for the partition's buffers.

The **CREATE_BUFFER** service creates an empty buffer with the following specified information:

- name, which must be unique within the partition
- maximum number of messages
- maximum message size
- discipline for queuing waiting processes (**FIFO** or **PRIORITY**)

The service returns a buffer ID.

Sending Messages to APEX Buffers

The **SEND_BUFFER** service sends a message to a specified buffer.

If the buffer is empty, the message is stored in FIFO order. If processes are waiting for messages, the first process is removed from the queue and put in the **READY** state.

If the buffer is full, the sending process is put in the **WAITING** state and put in the buffer's send queue according to the buffer's queuing discipline and the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Receiving Messages from APEX Buffers

The **RECEIVE_BUFFER** receives a message from a specified buffer.

If the buffer is not empty, the message is removed from the FIFO queue.

If the buffer is full and processes are waiting for messages, the first process is removed from the send queue and put in the **READY** state.

If the buffer is empty, the receiving process is put in the **WAITING** state and put in the receive queue according to the buffer's queuing discipline and the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Getting APEX Buffer Information

The **GET_BUFFER_ID** service gets the buffer ID of a specified buffer.

The **GET_BUFFER_STATUS** service gets the following information for the specified buffer:

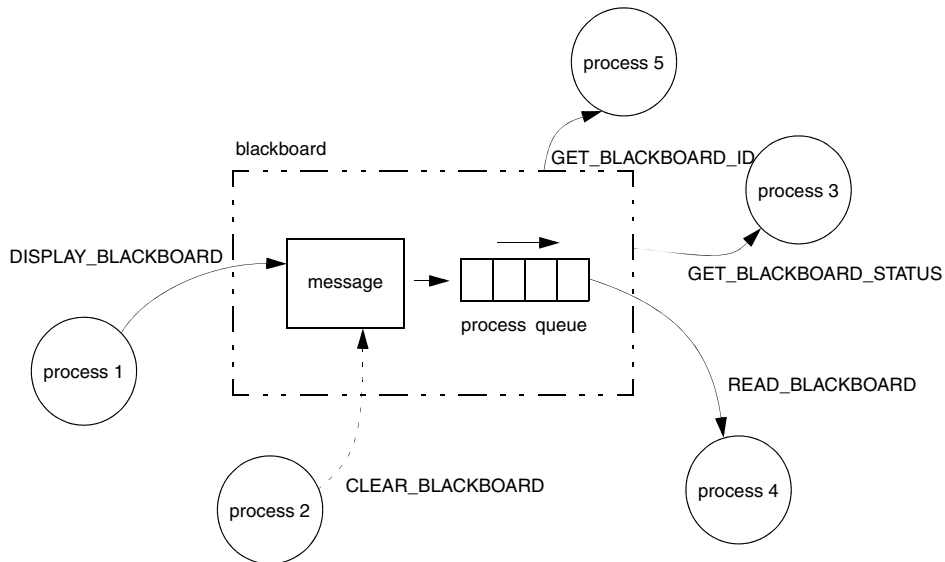
- current number of messages
- number of waiting processes
- maximum allowable number of messages
- maximum allowable message size

4.9.2 Communicating Using APEX Blackboards

APEX blackboards let processes communicate with each other within a partition. Blackboards support a single message type between multiple source and destination processes. Communication is indirect: participating processes address the blackboard rather than the opposing processes directly, thus providing a level of process independence.

Figure 4-9 summarizes how processes use a blackboard to communicate.

Figure 4-9 Processes Using a Blackboard to Communicate



Creating Blackboards

Blackboards can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many blackboards as are supported by the memory that is pre-allocated for the partition's blackboards.

The **CREATE_BLACKBOARD** service creates an empty blackboard with the following specified information:

- name, which must be unique within the partition
- maximum number of messages

The service returns a blackboard ID.

Displaying Blackboard Messages

The **DISPLAY_BLACKBOARD** service writes a message on a blackboard and remove all waiting processes from the process queue, putting them in the **READY** state. The message remains on the blackboard.

Reading Blackboard Messages

If the specified blackboard is not empty, the **READ_BLACKBOARD** service reads the displayed message from it.

If the blackboard is empty, the reading process is put in the **WAITING** state according to the specified timeout value.

If the service times out, it returns **TIMED_OUT**.

Clearing Blackboards

The **CLEAR_BLACKBOARD** service clears the message from the specified blackboard. As a result, the blackboard becomes empty.

Getting Blackboard Information

The **GET_BLACKBOARD_ID** service gets the blackboard ID for the specified name.

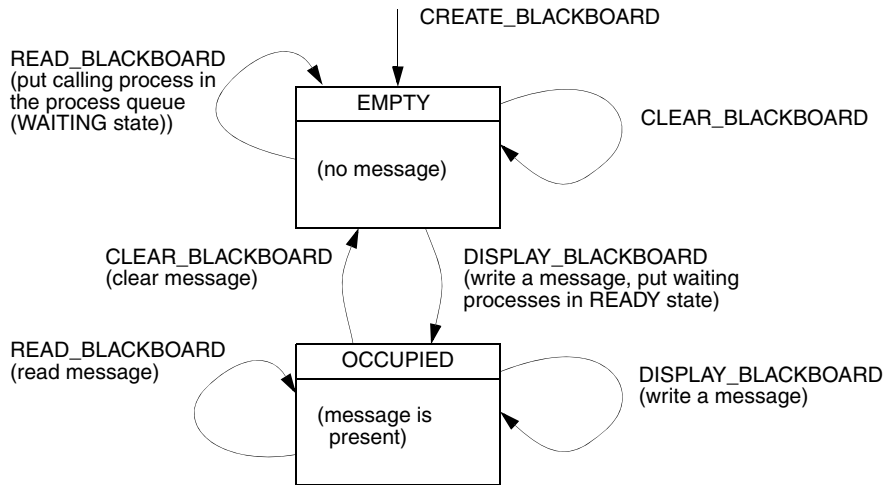
The **GET_BLACKBOARD_STATUS** service gets the following information for the specified blackboard:

- state (**EMPTY** or **OCCUPIED**)
- number of processes waiting for a message
- maximum allowable message size

State Transitions for Blackboards

[Figure 4-10](#) shows state transitions for blackboards.

Figure 4-10 State Transitions for Blackboards



4.9.3 Communicating Using APEX Semaphores

APEX semaphores are counting semaphores. A process waits on a semaphore to gain access to a resource and then signals the semaphore when it is finished. A semaphore's current value indicates the number of times that it is currently available to be taken.

For information on vThreads semaphores, see [A.3.3 Semaphores](#), p.296. For information on POSIX semaphores, see [5.6 POSIX Semaphores](#), p.101.

Creating APEX Semaphores

APEX semaphores can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many APEX semaphores as are supported by the memory that is pre-allocated for the partition's APEX semaphores.

The **CREATE_SEMAPHORE** service creates a semaphore with the following specified information:

- name, which must be unique within the partition
- maximum value
- current value

- queuing discipline (**FIFO** or **PRIORITY**)

The service returns a semaphore ID.

Waiting for APEX Semaphores

If the specified semaphore's current value is not zero, the **WAIT_SEMAPHORE** service decrements the value, and the process continues to run.

If the current value is zero, the process is put in the **WAITING** state and queued according to the semaphore's queuing discipline and the specified timeout.

If the service times out, it returns **TIMED_OUT**.

Signalling APEX Semaphores

If there are no processes waiting for the specified semaphore, the **SIGNAL_SEMAPHORE** service increments the semaphore's current value.

If there are processes waiting for the semaphore, the service uses the semaphore's queuing discipline to determine which process will receive the signal and sets that process's state to **READY**.

Getting APEX Semaphore Information

The **GET_SEMAPHORE_ID** service gets the semaphore ID for the specified name.

The **GET_SEMAPHORE_STATUS** service gets the following information for the specified semaphore:

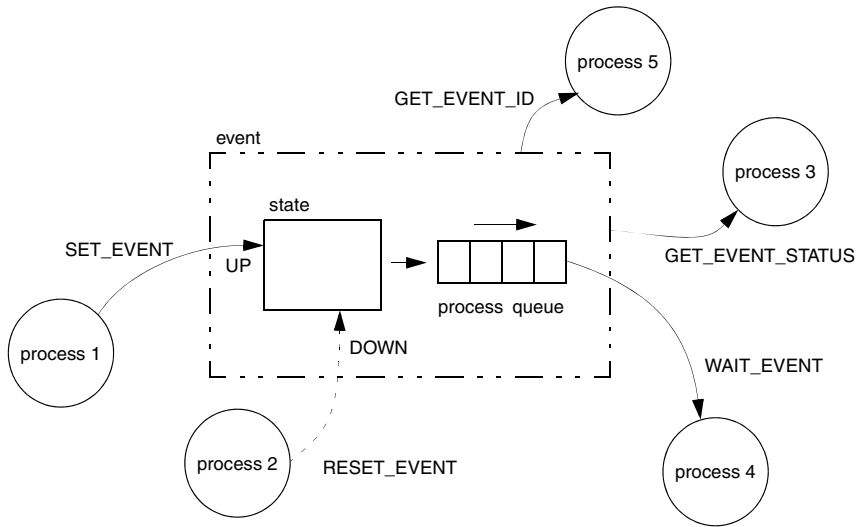
- current count
- number of processes waiting for the semaphore
- maximum value

4.9.4 Synchronizing Using APEX Events

APEX events let processes in a partition synchronize. Processes that are waiting for a condition are notified when the condition happens. An event can be in one of two states: **UP** or **DOWN**.

[Figure 4-11](#) summarizes how processes use an APEX event to synchronize.

Figure 4-11 Synchronizing Using an APEX Event



CAUTION: Processes should not count the occurrences of an event. Multiple **SET_EVENT** conditions that occur in a short period of time, or conditions that occur when no processes are waiting for the event, are coalesced into one **UP** state.

Event Queuing

Rescheduling of processes occurs when a process attempts to wait for an event that is down. The calling process is queued for a specified amount of time (the time can be infinite). If the event is not set (up) in that amount of time, VxWorks 653 automatically removes the process from the queue, sets the return code to **TIMED_OUT**, and puts the process back into the ready state.

Creating APEX Events

APEX events can be created only when a partition is being initialized, that is, when the partition mode is anything but **NORMAL**. Processes can create as many APEX events as are supported by the memory that is pre-allocated for the partition's APEX events.

The **CREATE_EVENT** service creates an event with a specified name, which must be unique within the partition. The service returns an event ID. The event starts in the **DOWN** state.

Setting and Resetting APEX Events

The **SET_EVENT** service sets the specified event to the **UP** state. All the processes waiting for the event are put into the **READY** state.

The **RESET_EVENT** service sets the specified event to the **DOWN** state.

Waiting for APEX Events

If the specified event is in the **DOWN** state, the **WAIT_EVENT** service moves the calling process from the **RUNNING** state to the **WAITING** state. If the event is in the **UP** state, the calling process continues to run.

Getting APEX Event Information

The **GET_EVENT_ID** service gets the event ID for the specified event.

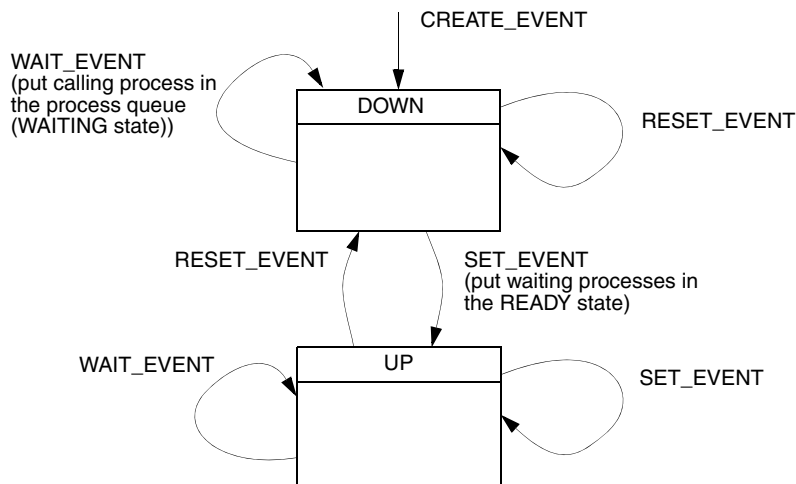
The **GET_EVENT_STATUS** service gets the following status information for the specified event:

- state (**UP** or **DOWN**)
- number of waiting processes

State Transitions for APEX Events

Figure 4-12 shows state transitions for APEX events.

Figure 4-12 State Transitions for APEX Events



4.10 Monitoring Health in APEX Partitions

APEX error services (in the **apexError** library) support process-level health monitoring as defined in the ARINC 653 standard.

When an APEX process raises an error, the partition's error handler process runs.

4.10.1 Raising Process-Level Errors

When an APEX partition detects an error, it issues the **RAISE_APPLICATION_ERROR** service with an error code and a fault message. The service causes the error handler process to run.

Depending on its nature and scope, an error raised at the process level with the **RAISE_APPLICATION_ERROR** service could propagate to the partition level, where it is processed.

4.10.2 APEX Errors

The following are process-level APEX error codes:

- **APPLICATION_ERROR**
- **DEADLINE_MISSED**
- **HARDWARE_FAULT**
- **ILLEGAL_REQUEST** (invalid or illegal OS call)
- **MEMORY_VIOLATION**
- **NUMERIC_ERROR**
- **POWER_FAIL**
- **STACK_OVERFLOW** (process stack overflow)

A faulty process can continue to run only in the cases of **APPLICATION_ERROR** or **DEADLINE_MISSED**.

4.10.3 Creating Error Handler Processes

An APEX application creates an error handler process for a partition by issuing the **CREATE_ERROR_HANDLER** service with the error handler entry point and stack size. The application supplies the error handler code.

The error handler is an aperiodic process that runs in the partition window with the highest priority of any process in the partition. It preempts any process regardless of its priority, even if preemption is disabled for the partition. It has no process ID and cannot be accessed by other processes within the partition. That is, other processes cannot suspend it, stop it, or change its priority.

An application developer writes the error handler. It could do a selection of the following:

- Get information about the error (issue **GET_ERROR_STATUS**):
 - error code (see [4.10.2 APEX Errors](#), p.86)
 - error address
 - process ID of the faulty process
 - fault message

If more than one process is faulty, the error handler process must issue **GET_ERROR_STATUS** in a loop until there are no more processes in error (that is, until the service returns **NO_ACTION**).
- Get information about the failed process (issue **GET_PROCESS_STATUS**).
- Stop (issue **STOP**) or restart (issue **START**) the failed process.
- Restart the partition (issue **SET_PARTITION_MODE** with **WARM_START** or **COLD_START**).
- Shut down the partition (issue **SET_PARTITION_MODE** with **IDLE**).
- Escalate the error to the partition level (issue **REPORT_APPLICATION_ERROR**).
- Log the fault message with the health monitor (issue **REPORT_APPLICATION_MESSAGE**).
- Stop itself (issue **STOP_SELF**).

If code running in the context of the error handler calls **LOCK_PREEMPTION** or **UNLOCK_PREEMPTION**, no action is taken. This is because the error handler is already the highest-priority process and cannot be interrupted or blocked. It can transmit the error context to health monitoring via the **REPORT_APPLICATION_MESSAGE** service for maintenance purpose.

An error handler process cannot do the following:

- Correct an error. For example, it cannot limit a value in the case of overflow.
- Call blocking services.

Errors that occur while the error handler process runs are partition-level errors.

5

Developing POSIX Applications

5.1	Introduction	89
5.2	POSIX Clocks and Timers	90
5.3	POSIX Memory-Locking Interface	91
5.4	POSIX Threads	92
5.5	POSIX Scheduling Interface	97
5.6	POSIX Semaphores	101
5.7	POSIX Mutexes and Condition Variables	108
5.8	POSIX Message Queues	109
5.9	POSIX Queued Signals	120
5.10	POSIX API for vThreads Partitions	121

5.1 Introduction

The POSIX standard for real-time extensions (1003.1b) specifies a set of interfaces to kernel facilities. To improve application portability, for vThreads partitions, VxWorks 653 provides POSIX interfaces as well as vThreads (both C and C++) and APEX.

This chapter discusses programming concepts for writing POSIX applications that run in vThreads partitions.

This chapter uses the qualifier *Wind* to identify facilities designed for use with layers other than the POSIX API. For example, you can find a discussion of Wind semaphores contrasted to POSIX semaphores in [5.6.1 Comparison of POSIX and Wind Semaphores](#), p. 102.

POSIX asynchronous I/O (AIO) routines are available in the **aioPxLib** library. The VxWorks 653 AIO implementation meets the specification in the POSIX 1003.1b standard.

5.2 POSIX Clocks and Timers

A clock is a software construct (**struct timespec**, defined in **time.h**) that keeps time in seconds and nanoseconds. The software clock is updated by system-clock ticks. VxWorks 653 provides a POSIX 1003.1b standard clock and timer interface.

The POSIX standard provides a means of identifying multiple virtual clocks, but only one clock is required: the VxWorks 653 module-wide real-time clock. No virtual clocks are supported in VxWorks 653.

The VxWorks 653 module-wide real-time clock is identified in the clock and timer routines as **CLOCK_REALTIME**, and is defined in **time.h**. VxWorks 653 provides routines to access the module-wide real-time clock. For more information, see the reference entry for **clockLib**.

The POSIX timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to create, set, and delete a timer. For more information, see the reference entry for **timerLib**. When a timer goes off, the default signal, **SIGALRM**, is sent to the task. To install a signal handler that runs when the timer expires, use **sigaction()** (see [A.3.6 Signals](#), p. 315).

Example 5-1 POSIX Timers

```
/* This example creates a new timer and stores it in timerid. */

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

int createTimer (void)
```

```
{
timer_t timerid;

/* create timer */
if (timer_create (CLOCK_REALTIME, NULL, &timerid) == ERROR)
{
    printf ("create FAILED\n");
    return (ERROR);
}
return (OK);
}
```

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

5.3 POSIX Memory-Locking Interface

To use more virtual memory than there is physical memory, many operating systems page and swap memory. This technique causes unpredictable delays in running time, so it is not desirable in real-time systems. Since VxWorks 653 is designed for real-time systems, it does not page or swap memory.

However, the POSIX 1003.1b standard for real-time extensions covers operating systems that do page and swap. Such systems that want real-time performance can use the POSIX page-locking facilities to declare that certain memory blocks must not be paged or swapped.

To increase portability between other POSIX-compliant systems and VxWorks 653, VxWorks 653 includes POSIX page-locking routines. Since all memory is always locked in a VxWorks 653 system, calling the routines has no effect.

The POSIX page-locking routines are in the **mmanPxLib**. The library name indicates the POSIX memory-management library. All routines return **OK** (0).

POSIX libraries are available when **INCLUDE_POSIX** is included in a vThreads partition. For detailed information about the libraries and their routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

5.4 POSIX Threads

POSIX threads (pThreads) are similar to vThreads tasks (called vThreads threads in VxWorks 653), but with some additional characteristics, including a thread ID that differs from its task ID.

5.4.1 pThread Attributes

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

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

Stack Size

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

- Attribute name: **stacksize**
- Default value: default stack size set for **taskLib**
- Access routines:
 - **pthread_attr_getstacksize()**
 - **pthread_attr_setstacksize()**

Stack Address

The stack address attribute specifies the base of a region of user-allocated memory to be used as a stack region for the created pThread. Because the default value is **NULL**, the VxWorks 653 module should allocate a stack for the pThread when it is created.

- Attribute name: **stackaddr**
- Default value: **NULL**
- Access routines:
 - **pthread_attr_getstackaddr()**

- `pthread_attr_setstackaddr()`

Detach State

The detach state attribute describes the state of a thread. With pThreads, the creator of a thread can block until the thread exits (see the entries for `pthread_exit()` and `pthread_join()` in the *VxWorks 653 vThreads API Reference*). In this case, the pThread is a joinable thread. Otherwise, it is a detached thread. A pThread that was created as a joinable thread can dynamically make itself a detached thread by calling `pthread_detach()`.

- Attribute name: **detachstate**
- Possible values:
 - `PTHREAD_CREATE_DETACHED`
 - `PTHREAD_CREATE_JOINABLE`
- Default value: `PTHREAD_CREATE_JOINABLE`
- Access routines:
 - `pthread_attr_getdetachstate()`
 - `pthread_attr_setdetachstate()`
- Dynamic access routine: `pthread_detach()`

Contention Scope

The contention scope attribute describes how pThreads compete for resources, namely the CPU. Under VxWorks 653, all threads compete for the CPU, so the competition is VxWorks 653 module-wide. Although POSIX allows two values, only `PTHREAD_SCOPE_SYSTEM` is implemented.

- Attribute name: **contentionscope**
- Possible values:
 - `PTHREAD_SCOPE_SYSTEM`
(`PTHREAD_SCOPE_PROCESS` is not implemented)
- Default value: `PTHREAD_SCOPE_SYSTEM`
- Access routines:
 - `pthread_attr_getscope()`

- `pthread_attr_setscope()`

Inherit Scheduling

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

- Attribute name: **inheritsched**
- Possible values:
 - `PTHREAD_EXPLICIT_SCHED`
 - `PTHREAD_INHERIT_SCHED`
- Default value: `PTHREAD_INHERIT_SCHED`
- Access routines:
 - `pthread_attr_getinheritsched()`
 - `pthread_attr_setinheritsched()`

Scheduling Policy

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

- Attribute name: **schedpolicy**
- Possible values:
 - `SCHED_FIFO` (priority-preemptive scheduling)
 - `SCHED_RR` (round-robin scheduling by priority)
- Default value: `SCHED_RR`
- Access routines:
 - `pthread_attr_getschedpolicy()`
 - `pthread_attr_setschedpolicy()`

Because the default value for the **inheritsched** attribute is `PTHREAD_INHERIT_SCHED`, the **schedpolicy** attribute is not used by default. For more information, see [5.5.3 Getting and Displaying the Current Scheduling Policy](#), p.100.

Scheduling Parameters

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

- Attribute name: **schedparam**
- Range of values: 0 – 255
- Default value: default task priority set for **taskLib**
- Access routines:
 - **pthread_attr_getschedparam()**
 - **pthread_attr_setschedparam()**
- Dynamic access routines:
 - **pthread_getschedparam()** using thread ID
 - **pthread_setschedparam()** using thread ID
 - **sched_getparam()** using task ID
 - **sched_setparam()** using task ID

Because the default value the **inheritsched** attribute is **PTHREAD_INHERIT_SCHED**, the **schedparam** attribute is not used by default. For more information, see [5.5.2 Getting and Setting POSIX Task Priorities](#), p.98.

Specifying Attributes when Creating pThreads

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

Example 5-2 Creating a pThread Using Explicit Scheduling Attributes

```
pthread_t tid;
pthread_attr_t attr;
int ret;
pthread_attr_init(&attr);

/* set the inheritsched attribute to explicit */
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

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

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

Example 5-3 **Creating a pThread Using Default Attributes**

```
pthread_t tid;
int ret;

/* create the pthread with NULL attributes to designate default values */
ret = pthread_create(&tid, NULL, entryFunction, entryArg);
```

Example 5-4 **Designating Your Own Stack for a pThread**

```
pthread_attr_init(&attr);

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

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

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

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

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

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

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

5.4.2 pThread Private Data

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

5.4.3 pThread Cancellation

POSIX provides a mechanism, called cancellation, to terminate a thread gracefully. There are two types of cancellation:

- Synchronous: Synchronous cancellation causes the pThread to explicitly check to determine if it was cancelled or to call a routine that contains a cancellation point.
- Asynchronous: Asynchronous cancellation causes the running of the pThread to be interrupted and a handler to be called, much like a signal. (Asynchronous cancellation is actually implemented with a special signal, **SIGCANCEL**, which applications should be careful not to block or ignore.)

Routines that can be used with cancellation are in **pthreadLib** and are listed in [Table 5-1](#). For more information, see reference entries in the *VxWorks 653 vThreads API Reference*.

Table 5-1 **pThreads Cancellation Routines**

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

5.5 POSIX Scheduling Interface

The POSIX 1003.1b scheduling routines are in **schedPxLib**. The routines let you use a portable interface to get the following:

- task priority (and set it)
- scheduling policy

- maximum and minimum priority for tasks
- length of a time slice if round-robin scheduling is in effect

For details about the library and its routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

This section describes how to use these routines, beginning with a list of the minor differences between the POSIX and Wind methods of scheduling.

5.5.1 Comparison of POSIX and Wind Scheduling

POSIX and Wind scheduling routines differ in the following ways:

- POSIX scheduling is based on processes. Wind scheduling is based on tasks.
- The POSIX standard uses the term *FIFO* scheduling. VxWorks 653 documentation uses the term priority-preemptive scheduling. Only the terms differ. Both describe the same priority-based policy.
- POSIX applies scheduling algorithms on a process-by-process basis. Wind applies scheduling algorithms on a partition-wide basis, meaning that all tasks in a partition use either a round-robin scheme or a priority-preemptive scheme.
- The POSIX priority numbering scheme is the inverse of the Wind scheme. In POSIX, the higher the number, the higher the priority. In the Wind scheme, the lower the number, the higher the priority, where 0 is the highest priority. Accordingly, the priority numbers used with the POSIX scheduling library, **schedPxLib**, do not match those used and reported by all other components of VxWorks 653. You can override this default by setting the global variable **posixPriorityNumbering** to **FALSE**. If you do this, **schedPxLib** uses the Wind numbering scheme (smaller number = higher priority) and its priority numbers match those used by the other components of VxWorks 653.

5.5.2 Getting and Setting POSIX Task Priorities

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

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

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

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

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

/* defines */
#define PX_NEW_PRIORITY 150

STATUS priorityTest (void)
{
    struct sched_param myParam;

    /* initialize param structure to desired priority */

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

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

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

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

    return (OK);
}
```

The `sched_setscheduler()` routine is designed to set both scheduling policy and priority for a single POSIX process, which corresponds in most other cases to a single Wind task. In the core OS, `sched_setscheduler()` controls only task priority,

because the kernel does not let tasks have scheduling policies that differ from one another. If its policy specification matches the current VxWorks 653 module-wide scheduling policy, **sched_setscheduler()** sets only the priority, thus acting like **sched_setparam()**. If its policy specification does not match the current one, **sched_setscheduler()** returns an error.

The only way to change the scheduling policy is to change it for all tasks. There is no POSIX routine for this purpose. To set a VxWorks 653 module-wide scheduling policy, use the Wind **kernelTimeSlice()** routine described in [Round-Robin Scheduling](#), p.274.

5.5.3 Getting and Displaying the Current Scheduling Policy

The **sched_getscheduler()** POSIX routine returns the current scheduling policy. There are two valid scheduling policies in VxWorks 653: priority-preemptive scheduling (in POSIX terms, **SCHED_FIFO**) and round-robin scheduling by priority (**SCHED_RR**). For more information, see [Scheduling Policy](#), p.94.

Example 5-6 Getting POSIX Scheduling Policy

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

/* includes */

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

STATUS schedulerTest (void)
{
    int policy;

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

    /* sched_getscheduler returns either SCHED_FIFO or SCHED_RR */

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

    return (OK);
}
```

5.5.4 Getting Scheduling Parameters: Priority Limits and Time Slice

The `sched_get_priority_max()` and `sched_get_priority_min()` routines return the maximum and minimum possible POSIX priority.

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

Example 5-7 Getting the POSIX Round-Robin Time Slice

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

/* includes */

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

STATUS rrgetintervalTest (void)
{
    struct timespec slice;

    /* turn on round robin */

    kernelTimeSlice (30);

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

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

5.6 POSIX Semaphores

POSIX defines both named and unnamed semaphores, which have the same properties, but use slightly different interfaces. The POSIX semaphore library

provides routines for creating, opening, and destroying both named and unnamed semaphores. When opening a named semaphore, you assign a symbolic name, which the other named-semaphore routines accept as an argument. The POSIX semaphore routines are in **semPxBLib**. For details about the library and its routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.



NOTE: Some host operating systems, such as UNIX, require symbolic names for objects that are to be shared among processes. This is because processes do not normally share memory in such operating systems. In VxWorks 653, there is no requirement for named semaphores, because all kernel objects have unique identifiers. However, using named semaphores of the POSIX variety provides a convenient way to determine the object's ID.

5.6.1 Comparison of POSIX and Wind Semaphores

POSIX semaphores are counting semaphores. That is, they keep track of the number of times they are given. The Wind semaphore mechanism is similar to that specified by POSIX, except that Wind semaphores offer additional features listed below:

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

The POSIX terms *wait* (or *lock*) and *post* (or *unlock*) correspond to the VxWorks 653 terms *take* and *give*. The POSIX routines for locking, unlocking, and getting the value of semaphores are used for both named and unnamed semaphores.

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

The **sem_open()**, **sem_unlink()**, and **sem_close()** routines 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.



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

5.6.2 Using Unnamed Semaphores

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

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

Semaphores can be used for both synchronization and exclusion.

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. Subsequent tasks block if the semaphore value was initialized to 1.

Example 5-8 POSIX Unnamed Semaphores

```
/* This example uses unnamed semaphores to synchronize an action between the
 * calling task and a task that it spawns (tSyncTask). To run from the shell,
 * spawn as a task:
 *      -> sp unnameSem
 */

/* includes */

#include "vxWorks.h"
#include "semaphore.h"

/* forward declarations */
void syncTask (sem_t * pSem);
```

```
void unnameSem (void)
{
    sem_t * pSem;

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

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

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

    /* do something useful to synchronize with syncTask */

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

    /* all done - destroy semaphore */
    if (sem_destroy (pSem) == -1)
    {
        printf ("unnameSem: sem_destroy failed\n");
        return;
    }
    free ((char *) pSem);
}

void syncTask
(
    sem_t * pSem
)
{
    /* wait for synchronization from unnameSem */
    if (sem_wait (pSem) == -1)
    {
        printf ("syncTask: sem_wait failed \n");
        return;
    }
}
```

```
else
    printf ("syncTask:sem locked; doing sync'ed action...\n");

/* do something useful here */
}
```

5.6.3 Using Named Semaphores

5

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 accessed semaphore already exists, are shown in [Table 5-2](#). There is no entry for **O_EXCL** alone, because using that flag alone is not meaningful.

Table 5-2 Possible Outcomes of Calling **sem_open()**

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

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

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

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

Example 5-9 POSIX Named Semaphores

```
/*
 * In this example, nameSem() creates a task for synchronization. The
 * new task, tSyncSemTask, blocks on the semaphore created in nameSem().
 * Once the synchronization takes place, both tasks close the semaphore,
 * and nameSem() unlinks it. To run this task from the shell, spawn
 * nameSem as a task:
 * -> sp nameSem, "myTest"
 */

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

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

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

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

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

    /* do something useful to synchronize with syncSemTask */

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

```

    }

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

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

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

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

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

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

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

    /* do something useful here */

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

```

5.7 POSIX Mutexes and Condition Variables

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

Mutex attributes are held in a data type called **pthread_mutexattr_t**, which contains two attributes, **protocol** and **prioceiling**.

Protocol

The protocol attribute describes how the mutex deals with the priority-inversion problem described in the section for mutual-exclusion semaphores ([Mutual-Exclusion Semaphores](#), p.302).

- Attribute name: **protocol**
- Possible values:
 - **PTHREAD_PRIO_INHERIT**
 - **PTHREAD_PRIO_PROTECT**
- Access routines:
 - **pthread_mutexattr_getprotocol()**
 - **pthread_mutexattr_setprotocol()**

To create a mutual-exclusion semaphore with priority inheritance, use the **SEM_Q_PRIORITY** and **SEM_PRIO_INHERIT** options to **semMCreate()**.

Mutual-exclusion semaphores created with the priority protection value use the notion of a priority ceiling, which is the other mutex attribute.

Priority Ceiling

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

- Attribute name: **prioceiling**
- Possible values: any valid (POSIX) priority value
- Access routines:
 - **pthread_mutexattr_getprioceiling()**
 - **pthread_mutexattr_setprioceiling()**

- Dynamic access routines:
 - `pthread_mutex_getprioceiling()`
 - `pthread_mutex_setprioceiling()`



NOTE: The POSIX priority numbering scheme is the inverse of the Wind scheme. See [5.5.1 Comparison of POSIX and Wind Scheduling](#), p.98.

A priority ceiling is defined by the following conditions:

- Any thread attempting to acquire a mutex, whose priority is higher than the ceiling, cannot acquire the mutex.
- Any thread whose priority is lower than the ceiling value has its priority elevated to the ceiling value for the duration that the mutex is held.
- The thread's priority is restored to its previous value when the mutex is released.

5.8 POSIX Message Queues

The POSIX message queue routines are in **mqPxBLib**. For details on the library and its routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

5.8.1 Comparison of POSIX and Wind Message Queues

The POSIX message queues are similar to Wind message queues, except that POSIX message queues provide messages with a range of priorities. The differences between the POSIX and Wind message queues are summarized in [Table 5-3](#).

Table 5-3 Message Queue Feature Comparison

Feature	Wind Message Queues	POSIX Message Queues
Blocked task queues	FIFO or priority-based	Priority-based
Close and unlink semantics	No	Yes

Table 5-3 **Message Queue Feature Comparison** (cont'd)

Feature	Wind Message Queues	POSIX Message Queues
Message priority levels	1	32
Receive with timeout	Optional	Not available
Task notification	Not available	Optional (one task)

POSIX message queues are also portable, if you are migrating to VxWorks 653 from another 1003.1b-compliant system.

5.8.2 POSIX Message Queue Attributes

A POSIX message queue has the following attributes:

- an optional `O_NONBLOCK` flag
- the maximum number of messages in the message queue
- the maximum message size
- the number of messages on the queue

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

Example 5-10 **Setting and Getting Message-Queue Attributes**

```
/*
 * This example sets the O_NONBLOCK flag and examines message queue
 * attributes.
 */

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

/* defines */
#define MSG_SIZE    16

int attrEx
(
    char * name
)
{
    mqd_t          mqPXId;          /* mq descriptor */
}
```



```

struct mq_attr  attr;                /* queue attribute structure */
struct mq_attr  oldAttr;             /* old queue attributes */
char            buffer[MSG_SIZE];
int             prio;

/* create read write queue that is blocking */
attr.mq_flags = 0;
attr.mq_maxmsg = 1;
attr.mq_msgsize = 16;
if ((mqPId = 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 (mqPId, &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 (mqPId, 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 (mqPId, &oldAttr) == -1)
    return (ERROR);
else
{
    /* test that we got the values we think we should */
    if (!(oldAttr.mq_flags & O_NONBLOCK) || (oldAttr.mq_curmsgs != 0))
        return (ERROR);
    else
        printf ("queue attributes are:\n\tblocking is %s\n\t
            message size is: %d\n\t
            max messages in queue: %d\n\t
            no. of current msgs in queue: %d\n",
            oldAttr.mq_flags & O_NONBLOCK ? "on" : "off",
            oldAttr.mq_msgsize, oldAttr.mq_maxmsg,
            oldAttr.mq_curmsgs);
}
}

```

```
/* clean up - close and unlink mq */  
if (mq_unlink (name) == -1)  
    return (ERROR);  
if (mq_close (mqPXiD) == -1)  
    return (ERROR);  
return (OK);  
}
```

5.8.3 Displaying Message-Queue Attributes

The VxWorks 653 **show()** command produces a display of the key message-queue attributes, for either POSIX or Wind message queues. POSIX libraries are available when **INCLUDE_POSIX** is included in a vThreads partition. For detailed information about the libraries and their routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

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

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

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

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

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

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



NOTE: The built-in **show()** routine handles Wind message queues. You can also use the Wind River Workbench inspector to get information on Wind message queues.

5.8.4 Communicating through a Message Queue

Before a set of tasks can communicate through a POSIX message queue, one of the tasks must create the message queue by calling **mq_open()** with the **O_CREAT** flag

set. Once a message queue is created, other tasks can open that queue by name to send and receive messages on it. Only the first task opens the queue with the **O_CREAT** flag. Subsequent tasks can open the queue for receiving only (**O_RDONLY**), sending only (**O_WRONLY**), or both sending and receiving (**O_RDWR**).

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

One of the arguments to **mq_send()** specifies a message priority. Priorities range from 0 (lowest priority) to 31 (highest priority). See [5.5.1 Comparison of POSIX and Wind Scheduling](#), p.98.

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

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

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

Example 5-11 POSIX Message Queues

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

/* mqEx.h - message example header */

/* defines */
#define MQ_NAME "exampleMessageQueue"

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

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

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

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

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

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

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

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

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

```

    }
}

/* sendTask.c - mq sending example */

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

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

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

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

    if ((mqPId = 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 (mqPId, MSG, sizeof (MSG), HI_PRIO) == -1)
    {
        printf ("sendTask: mq_send failed\n");
        return;
    }
    else
        printf ("sendTask: mq_send succeeded\n");
}

```

5.8.5 Notifying a Task That a Message Is Waiting

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

The **mq_notify()** routine specifies a signal to be sent to the task when a message is placed on an empty queue. This mechanism uses the POSIX data-carrying extension to signaling, which lets you, for example, carry a queue identifier with the signal (see [5.9 POSIX Queued Signals](#), p.120).

The **mq_notify()** routine is designed to alert the task only for new messages that are actually available. If the message queue already contains messages, no notification is sent when more messages arrive. If another task is blocked on the

queue with **mq_receive()**, that other task unblocks, and no notification is sent to the task registered with **mq_notify()**.

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

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

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

Example 5-12 Notifying a Task That a Message Is Waiting

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

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

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

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

/*
 * exMqNotify - example of how to use mq_notify()
 */

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

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

```

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

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

    /*
     * Install signal handler for the notify signal and fill in
     * a sigaction structure and pass it to sigaction(). Because the handler
     * needs the siginfo structure as an argument, the SA_SIGINFO flag is
     * set in sa_flags.
     */

    mySigAction.sa_sigaction = exNotificationHandle;
    mySigAction.sa_flags     = SA_SIGINFO;
    sigemptyset (&mySigAction.sa_mask);

    if (sigaction (SIGUSR1, &mySigAction, NULL) == -1)
    {
        printf ("sigaction failed\n");
        return (-1);
    }

    /*
     * Create a message queue - fill in a mq_attr structure with the
     * size and no. of messages required, and pass it to mq_open().
     */

    attr.mq_flags = O_NONBLOCK;                /* make nonblocking */
    attr.mq_maxmsg = 2;
    attr.mq_msgsize = MSG_SIZE;

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

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

```

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

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

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

exMqRead (exMqId);

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

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

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

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

return (0);
}
```



```

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

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

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

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

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

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

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

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

static void exMqRead
(
    mqd_t      exMqId
)
{
    char        msg[MSG_SIZE];
    int         prio;

```

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

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

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

5.9 POSIX Queued Signals

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

kill()

The **kill()** routine includes an application-specified value that is sent as part of the signal. You can use this value to supply whatever context your signal handler finds useful. This value is of type **sigval** (defined in **signal.h**). The signal handler finds it in the **si_value** field of one of its arguments, a **siginfo_t** structure. An extension to the POSIX **sigaction()** routine registers signal handlers that accept this additional argument.

sigqueue()

The **sigqueue()** routine enables the queuing of multiple signals for any task. The **kill()** routine, by contrast, delivers a single signal, even if multiple signals arrive before the handler runs.

VxWorks 653 includes seven signals reserved for application use, numbered consecutively from **SIGRTMIN**. The presence of these reserved signals is required by POSIX 1003.1b, but the specific signal values are not. For portability, specify these signals as offsets from **SIGRTMIN** (for example, write **SIGRTMIN+2** to refer to the third reserved signal number). All signals delivered with **sigqueue()** are

queued in numeric order, with lower-numbered signals queuing ahead of higher-numbered signals.

POSIX 1003.1b also includes an alternative means of receiving signals. The **sigwaitinfo()** routine differs from **sigsuspend()** or **pause()** in that it lets your application 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 call a signal handler even if one is registered. The **sigtimedwait()** routine is similar, except that it can time out.

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

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

5.10 POSIX API for vThreads Partitions

POSIX libraries are available when **INCLUDE_POSIX** is included in a vThreads partition. For detailed information about the libraries and their routines, see their reference entries in the *VxWorks 653 vThreads API Reference*.

6

Developing C++ Applications

- 6.1 Introduction 123
- 6.2 Configuring vThreads to Use C++ 124
- 6.3 Writing C++ Applications 124
- 6.4 Using C++ Libraries 127
- 6.5 Writing C++ Cert Applications 128

6.1 Introduction

You can develop C++ applications that run in full vThreads partitions. You can also develop applications with a subset of C++ that lets safety-critical applications be certified to Level A of the RTCA/DO-178B avionics software guidelines.

This documentation calls this subset the C++ cert subset and applications based on it C++ cert applications. It uses other terms as they are used by the C++ standard.

This documentation describes how to develop C++ applications that run in full vThreads partitions and includes additional information when developing C++ cert applications differs.

6.2 Configuring vThreads to Use C++

6.2.1 Specifying Additional Sections for Loading

The GNU C++ compiler generates special ELF sections that are not loaded by default. As a result, these sections need to be accounted for in the XML configuration file. This is done by specifying an **AdditionalSection** attribute in **MemorySize** elements. For details, see the configuration information in the *VxWorks 653 Configuration and Build Guide*.

6.2.2 Adding C++ Support to vThreads

By default, a vThreads partition does not support C++. You can add C++ support for full C++ or C++ cert applications. For details, see the build information in the *VxWorks 653 Configuration and Build Guide*.

6.2.3 Demangling C++ Symbol Names in the Target Shell

If you use the C++ demangler, symbol table queries in a vThreads target shell return human-readable (demangled) forms of C++ symbol names.

6.3 Writing C++ Applications

This section describes how to write C++ applications in general. For additional information specific to writing C++ cert applications, see [6.5 Writing C++ Cert Applications](#), p.128.

For information on the GNU C++ toolchain, see *Using the GNU Compiler Collection*.

6.3.1 Making C Symbols Accessible to C++ Code

To make C symbols accessible to C++ code, use the **extern "C"** syntax:

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

Symbols in VxWorks 653 are automatically available to C++, because the VxWorks 653 header files use this mechanism for declarations

6

Making C++ Symbols Accessible to C code

To reference a (non-overloaded, global) C++ symbol from C code, use the **extern "C"** syntax.

6.3.2 Adding Floating-Point Support to Tasks

Any vThreads task that uses C++ code must be spawned with the **VX_FP_TASK** option. Failure to use the option can result in hard-to-debug, unpredictable run-time corruption of floating-point registers.

6.3.3 Handling Exceptions

Since the C++ cert subset does not support exception handling (catch and throw), this section applies to full C++ only.

Turning off Exception Handling

By default, the GNU C++ compiler enables exception handling. To turn the feature off, use the **-fno-exceptions** compiler option.

Using the Pre-Exception Model of C++ Compilation

You can write code according to the pre-exception model of C++ compilation. For example, calls to **new** can check the returned pointer for a failure value of zero.

However, if you are concerned that exception-handling enhancements will not compile correctly, follow these guidelines:

- Use **new (nothrow)**.
- Do not explicitly turn on exceptions in your iostream objects.

GNU iostream does not throw unless **IO_THROW** is defined when the library is built and exceptions are explicitly enabled for the particular iostream object in use. The default is no exceptions. Exceptions have to be explicitly turned on for each **iostate** flag that needs to throw.

- Do not use string objects or, if you must, wrap them in blocks of:

```
try { } catch (...) { }
```

The Standard Template Library (STL) does not throw except in some methods in the **basic_string** class (of which “string” is a specialization).

Installing Your Own Termination Handler

As specified by the ANSI C++ standard, unhandled exceptions ultimately call **terminate()**. This routine suspends the offending task and sends a warning message to the console. You can modify this behavior by installing your own termination handler. To do so, call **set_terminate()**, which is defined in the **exception** header file.

6.3.4 Using Namespaces

You can use namespaces for your own code, according to the C++ standard. If you use the **std** namespace syntax, identifiers in the namespace are global. Therefore, they must be globally unique.

6.3.5 Disabling Run Time Type Information (RTTI)

For full C++, the GNU C++ compiler enables RTTI by default. The feature adds a small overhead to any C++ program that contains classes with virtual functions. To turn off the feature, use the **-fno-rtti** compiler option.

The C++ cert subset does not support RTTI.

6.3.6 Constructors and Destructors

Global constructors are called when a partition starts. They are called again when the partition is restarted (warm or cold). Destructors are not called, because the C++ objects are not retained through the restart process.

6.4 Using C++ Libraries

Since the C++ cert subset does not support C++ libraries, this section applies to full C++ only.



WARNING: Do not use the **-nostdinc** compiler option if you are using the **iostream** library or Standard Template Library (STL).

If you do, you will get warnings about missing header files. This is because, without the option, the compiler includes the directories containing the header files that are needed for the **iostream** library or STL.

6.4.1 Using the **iostream** Library

To use the **iostream** library, include one or more of its header files after the **vxWorks.h** header file in the appropriate modules of your application. The most frequently used header file is **iostream**, but others are available. For information, see the C++ reference entries.

Standard **iostream** Objects

The standard **iostream** objects (**cin**, **cout**, **cerr**, and **clog**) are global. That is, they are not private to any particular vThreads task. They are correctly initialized regardless of the number of tasks or modules that reference them, and they can safely be used across multiple tasks that have the same definitions of **stdin**, **stdout**, and **stderr**. However, they cannot safely be used when different tasks have different standard I/O file descriptors. In such cases, the application is responsible for handling mutual exclusion.

Simulating Private Standard `iostream` Objects

To simulate private standard `iostream` objects, create a new `iostream` object of the same class as the standard `iostream` object (for example, `cin` is an `istream_withassign`), and assign to it a new `filebuf` object tied to the appropriate file descriptor. The new `filebuf` and `iostream` objects are private to the calling task, ensuring that other tasks cannot corrupt them.

```
ostream my_out (new filebuf (1));          /* 1 == STDOUT */
istream my_in (new filebuf (0), &my_out); /* 0 == STDIN;
                                           * TIE to my_out */
```

6.4.2 Using Standard Template Library (STL)

The GNU port of the STL for `vThreads` is thread-safe at the class level. This means that two tasks in the same domain can safely reference the same class-level data at the same time. However, the STL is not thread-safe at the object level. That is, if two tasks need to reference the same object at the same time, they must use a mutex semaphore.

You can use the STL in client code that is compiled with exception handling turned off. In `vThreads`, this means the following:

- For all checks that the caller can reasonably make (such as bounds checking), no action is taken where an exception would be thrown. Optimization being on is equivalent to removing these checks.
- If you are using default allocators and memory exhaustion occurs where `bad_alloc` would be thrown, the following message is logged (if logging is included):

```
"STL Memory allocation failed and exceptions disabled -calling terminate"
and the task calls terminate(). However, you can define custom allocators that
behave differently.
```

6.5 Writing C++ Cert Applications

In addition to the information on writing C++ applications that is described elsewhere in this documentation, this section is specific to writing C++ cert applications.

6.5.1 Features Not Supported

To ensure certifiability, the C++ cert subset does not support the following C++ features:

- C++ standard library
- exception handling (catch and throw)
- pure virtual functions (virtual functions are supported)
- RTTI
- STL

6.5.2 Persistent Global Constructors

Regular global constructors are called when a partition starts (warm or cold starts). For the C++ cert subset, persistent global constructors are called during partition cold start, but not during partition warm restart.

Specifying Persistent Global Constructors in Makefiles

The munch facility puts regular and persistent global constructors in separate data sections. For persistent global constructors only, you must specify them with the **-persistent-def** option. The following is an example of a makefile rule for handling C++ global persistent constructors:

```
MUNCHFLAGS_EXTRA = -persistent-def myPersistentConstructorsFile
```

Allocating Persistent Global Constructors

Applications are responsible for allocating persistent global constructors from persistent memory. This can be done in any of the following ways:

- allocating from a pool of persistent objects
- allocating from persistent heap
- using linker scripts

Allocating from a Pool of Persistent Objects

The follow code is an example of how to allocate pools of persistent objects.

```
#define MAX_POOL 512
int objectPool[MAX_POOL] __attribute__((__section__(".persistent.bss")));
int objectPoolCount __attribute__((__section__(".persistent.data"))) = 0;
```

The follow code is an example of a persistent constructor allocating an object from the storage:

```
if (objectPoolCount < MAX_POOL)
    myObject = &objectPool[objectPoolCount++];
```

Allocating from Persistent Heap

To have a persistent heap, an application needs to create a memory pool in the persistent BSS section. The following code is an example of how to do this:

```
#define HEAP_SIZE (64*1024)
char persistentHeap[HEAP_SIZE]
__attribute__((__section__(".persistent.bss")));
PART_ID persistentHeapId __attribute__((__section__(".persistent.bss")));
BOOL persistentHeapInitialized
__attribute__((__section__(".persistent.data"))) = FALSE;
```

The follow example code excerpt is from an allocation routine for the persistent heap:

```
if (!persistentHeapInitialized)
    memPartInit (&persistentHeapId, persistentHeap, HEAP_SIZE);
return memPartAlloc (&persistentHeapId, bytes_wanted)
```

Using Linker Scripts

A linker script can be used to place variables into **.persistent.bss** and **.persistent.data** sections.

6.5.3 Calling Pure Virtual Functions

The C++ cert subset catches erroneously called pure virtual functions, such as ones called because of an application run-time defect. If a pure virtual function is called from a C++ cert application, a health monitor event (code **VX_ERROR_KERNEL**, subcode **VX_ERR_NO_SUB_CODE**) is injected and the offending task is suspended.

6.5.4 Deallocating Heap

If a C++ cert application tries to deallocate heap by calling **free()** (directly or indirectly from other routines that the application calls), a health monitor event (code **VX_ERROR_KERNEL**, subcode **VX_ERR_UNHANDLED_EVENT**) is injected.

The memory is not freed. The offending task is suspended unless the health monitor is configured not to suspend offending tasks for this type of event.

7

Programming in the Core OS

- 7.1 Introduction 134
- 7.2 Partitions 135
- 7.3 VxWorks 653 Stacks 141
- 7.4 Shared Libraries 142
- 7.5 Shared Data Regions 143
- 7.6 User Configuration Records 147
- 7.7 Multitasking 147
- 7.8 Managing Memory 147
- 7.9 Restart Functionality 156
- 7.10 Partition Support 165
- 7.11 Worker Tasks 168
- 7.12 System Time 169
- 7.13 Partition Scheduling 169
- 7.14 Design Models for Ports 181
- 7.15 Setting up Communication with Other Modules 185

7.1 Introduction

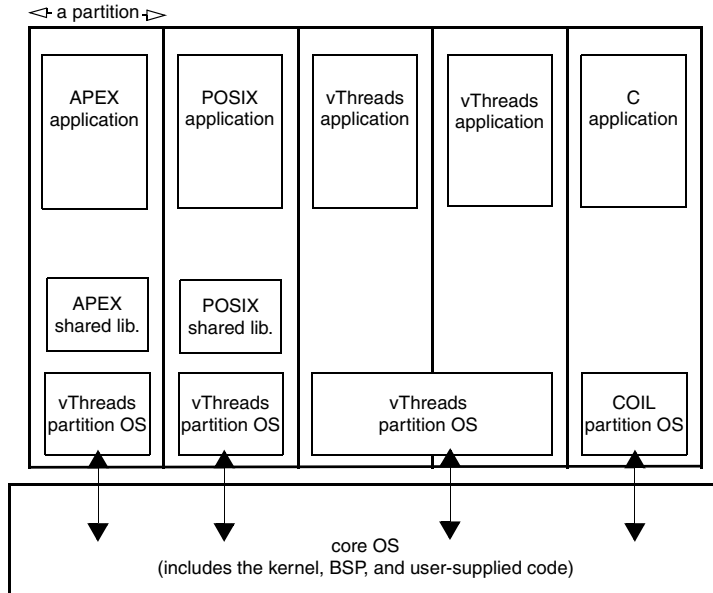
The core OS consists of the kernel, the BSP, and user-supplied code that runs in supervisor mode.

The following elements can be part of a VxWorks 653 module managed by the core OS:

- partitions
- shared libraries
- shared data regions
- online-loaded partitions

A configuration might resemble [Figure 7-1](#), which shows five partitions, and two (vThreads) of them sharing the same partition OS. For shared libraries and partition OSs, the figure shows relative virtual addresses within a partition, not within the module.

Figure 7-1 **Sample Configuration with Five Partitions**



7.2 Partitions

Partitions provide usage isolation of system resources for applications running in a VxWorks 653 module. Resources are CPU time, memory, and I/O. Each partition is allowed to use as much of these resources as is specified in the system configuration. A partition cannot access memory reserved for another partition, and it can run tasks only during its allocated time slot. For more information see [7.12 System Time](#), p.169.

Partitions are implemented as application domains. Only partitions can be application domains. A partition can run code that is included directly in the application domain, or code in attached shared libraries. Only partition OS components and application code can be included in partitions. Each such domain runs exactly one user-mode core OS task, running an instance of the partition OS. The partition OS can be either a vThreads partition OS (or a certifiable subset of it) or a partition OS based on COIL. For information on the vThreads partition OS, see [2. Developing vThreads Applications](#). For information on COIL, see [3. Developing COIL Applications](#).

7.2.1 Partition Configuration

Partitions are created at system startup according to their configuration in the XML configuration file. The configuration for vThreads and COIL partitions is similar. For details, see the *VxWorks 653 Configuration and Build Guide*.

To get the partition configuration information at run-time from the core OS, call **configRecordFieldGet()** with **PARTITION_CFG_RECORD** record and the appropriate field selector as shown below. The same routine can be called from a vThreads partition, but the record type is not needed. As noted below, some information is not available from a vThreads partition.

PARTITION_ALLOC_DISABLE

When set to **TRUE**, dynamically allocating and freeing partition heap memory is disabled when **NORMAL** operation mode is set with **PARTITION_MODE_SET()** or explicitly by calling **memPartAllocDisable()**.

PARTITION_CRITICALITY

The RTCA/DO-178B criticality level of the partition.

PARTITION_DURATION

Amount of processor time, in **SYSTEM_TIME_TYPE** increments, given to a partition every period. It is used to validate schedules. If set to **ZERO_TIME_VALUE**, a duration is not specified.

PARTITION_EVENTQ_STALL_DURATION

Maximum time allowed, in **SYSTEM_TIME_TYPE** units, for a pseudo-interrupt event to remain in the pseudo-interrupt event queue. The value is used to detect stalled partitions. The associated timer runs only when the partition is active and is, therefore, immune to changes in schedule. If set to **INFINITE_TIME_VALUE**, the feature is disabled.

PARTITION_FP_EXC_ENABLE

When set to **TRUE**, support for floating-point exceptions is enabled in the vThreads partition.

PARTITION_HM

Pointer to the partition's health monitor table. For details, see [8.5 Getting Health Monitor Information at Run-time](#), p.215. This information is not available from a vThreads partition.

PARTITION_ISR_STACK_SIZE

Size, in bytes, of the partition's interrupt stack. If set to **NONE**, the size is the value specified for the partition OS component; for a vThreads partition, the value is **ISR_STACK_SIZE**.

PARTITION_MAX_FDS

Maximum number of global file descriptors that can be opened by the partition.

PARTITION_NAME

Partition name, from 1 to **MAX_NAME_LENGTH** (as specified in **apexType.h**) characters in a NULL-terminated ASCII string.

PARTITION_NUM_DRIVERS

Maximum number of I/O device drivers in the partition OS. If set to **NONE**, the maximum number is the value specified for the partition OS component; for a vThreads partition, the value is **NUM_DRIVERS**.

PARTITION_NUM_FILES

Maximum number of open files allowed in the partition, including the number of global file descriptors. For example, if the partition opens 130 channels, the number of global file descriptors must be at least 130 and

PARTITION_NUM_FILES must be at least 130. If set to **NONE**, the maximum number is the value specified for the partition OS component. For a vThreads partition, the value is **NUM_FILES**.

PARTITION_NUM_LOG_MSGS

Maximum number of messages allowed in the logging queue. If set to **NONE**, the maximum number is the value specified for the partition OS component. For a vThreads partition, the value is **MAX_LOG_MSGS**.

PARTITION_NUM_SD_RGNS

Number of shared data regions in the **PARTITION_SD_RGN_NAME** field. This information is not available from a vThreads partition.

PARTITION_NUM_STK_GUARD_PAGES

The number of stack guard pages at the end of each vThreads task's stack. The partition OS uses the value to detect interrupt stack overflow and task stack overflow. If set to **NONE**, the number is the value specified for the partition OS component; for a vThreads partition, the value is

NUM_STACK_GUARD_PAGES.

PARTITION_NUM_WORKER_TASKS

Number of worker tasks that the core OS provides for the partition. The value should be zero, except to support some Wind River tools, such as the partition shell, which requires two worker tasks for the partition in which it is enabled.

PARTITION_NUMBER

A number from 1 to **MAX_NUMBER_OF_PARTITIONS** (as specified in **apexType.h**) that uniquely identifies the partition. It is used in the schedule and port configuration records. By convention, the first partition is **partition 1**.

PARTITION_PERIOD

Activation period, in **SYSTEM_TIME_TYPE** increments, of the partition. It is used to validate schedules. If set to **ZERO_TIME_VALUE**, a period is not specified.

PARTITION_PPS_SCHED_CFG

PPS scheduling parameters. For details, see *PPS Scheduling Parameters*, p. 140. This information is not available from a vThreads partition.

PARTITION_SC_PERMISSION

Bitmask that determines the set of system calls a partition is allowed to perform. For details, see *System Call Permission Bitmasks*, p. 138.

PARTITION_SD_RGN_NAME

NULL or the address of the array of shared data region names to attach (that is, give access) to the partition. This information is not available from a vThreads partition.

PARTITION_SELECT_SERVER_QSIZE

Maximum number of concurrent vThreads tasks allowed to do a select operation on global file descriptors. If set to **NONE**, the maximum number is the value specified for the partition OS component; for a vThreads partition, the value is **SELECT_SERVER_QSIZE**.

PARTITION_USER1

For user-specified extension.

PARTITION_USER2

For user-specified extension.

PARTITION_WD_DURATION

Maximum partition time, in **SYSTEM_TIME_TYPE** increments, that an application can lock preemption while protecting critical sections. If set to **ZERO_TIME_VALUE**, the watchdog is disabled.

System Call Permission Bitmasks

The system call permission field in a partition configuration record (**PARTITION_SC_PERMISSION** selector) is the logical OR of various permission bitmasks. Interpreting the field is described in the following sections.

If default permissions are set (that is, no permission to make any system calls), the field is equal to the **SYSCALL_DEFAULT_PERMISSION** group bitmask.

If all permissions are set, the field is equal to the **SYSCALL_ALL_PERMISSION** group bitmask.



NOTE: If you use an individual bitmask rather than a group bitmask to determine the value of the field, do so as follows:

SYSCALL_SHIFT (*individualBitmask*)

I/O Permission Bitmasks

If all I/O permissions are set, the field is equal to the **SYSCALL_IORW_PERMISSION** group bitmask, which is the logical OR of the bitmasks below.

If just read I/O permissions are set, the field is equal to the **SYSCALL_IOR_PERMISSION** group bitmask, which is the logical OR of all the following bitmasks except the ones for write, ioctl, and create.

Individual I/O permission bitmasks are (called with **SYSCALL_SHIFT()**):

- **SYSCALL_IO_CLOSE**
- **SYSCALL_IO_CREAT**
- **SYSCALL_IO_DEVICE_FIND**
- **SYSCALL_IO_IOCTL**
- **SYSCALL_IO_OPEN**
- **SYSCALL_IO_READ**

- `SYSCALL_IO_REMOVE`
- `SYSCALL_IO_SELECT`
- `SYSCALL_IO_UNSELECT`
- `SYSCALL_IO_WRITE`

Port Permission Bitmasks

If all port permissions are set, the field is equal to the `SYSCALL_PORT_PERMISSION` group bitmask, which is the logical OR of the following individual bitmasks (called with `SYSCALL_SHIFT()`):

- `SYSCALL_PORT_ATTACH`
- `SYSCALL_PORT_DETACH`
- `SYSCALL_PORT_INT_RECV`
- `SYSCALL_PORT_INT_SEND`
- `SYSCALL_PORT_SEND`
- `SYSCALL_PORT_STATUS`

Message Queue Permission Bitmasks

If all message queue permissions are set, the field is equal to the `SYSCALL_MSGQ_PERMISSION` group bitmask, which is the logical OR of the following individual bitmasks (called with `SYSCALL_SHIFT()`):

- `SYSCALL_MSGQ_CLOSE`
- `SYSCALL_MSGQ_OPEN`
- `SYSCALL_MSGQ_RECV`
- `SYSCALL_MSGQ_SEND`

Scheduler Permission Bitmask

If all scheduler permissions are set, the field is equal to the `SYSCALL_SCHEDULER_PERMISSION` group bitmask. Since there is only one scheduler permission, this group bitmask is equivalent to (called with `SYSCALL_SHIFT()`):

- `SYSCALL_SCHEDULER_MODE_SET`

PPS Scheduling Bitmasks

If all PPS scheduling permissions are set, the field is equal to the `SYSCALL_PPS_SET_MY_PRIORITY_PERMISSION` group bitmask. Since there is

only one permission in this group, it is equivalent to (called with **SYSCALL_SHIFT()**):

- **SYSCALL_PPS_MY_SCHED**

If all PPS scheduling permissions are set for “my” partition, the field is equal to the **SYSCALL_PPS_SET_PRIORITY_PERMISSION** group bitmask. Since there is only one permission in this group, it is equivalent to (called with **SYSCALL_SHIFT()**):

- **SYSCALL_PPS_SCHED**

Custom Permission Bitmask

If all custom permissions are set, the field is equal to the **SYSCALL_CUSTOM_PERMISSION** group bitmask. Since there is only one custom permission, this group bitmask is equivalent to (called with **SYSCALL_SHIFT()**):

- **SYSCALL_CUSTOM**

PPS Scheduling Parameters

PPS scheduling parameters are defined by the following structure:

```
typedef struct pps_cfg_record
{
    CONFIGURATION_RECORD_TYPE type;
    int crSize;
    BOOL appsIdleRelinquishEnabled;
    int appsPriority;
} PPS_CFG_RECORD;
```

type

Type of configuration record; always **CFG_TYPE_PPS**.

crSize

Number of bytes in the structure.

appsIdleRelinquishEnabled

When set to **TRUE**, the partition is willing to relinquish its remaining partition window when the partition is determined to be idle or when the application forces the idle condition.

appsPriority

A value of -1 disables PPS scheduling for the partition. Values of 0 to 255 are valid partition priorities and indicate that the partition should be considered for PPS scheduling. Zero is the highest priority.

7.3 VxWorks 653 Stacks

System Call Stacks

VxWorks 653 supports one system call stack per partition. The system call stack services a partition's system call handler, which runs when a partition thread makes a system call request of the kernel. As a result, system call handlers run in the context of the core OS, thus providing better robustness and security compared to running on the partition task stack.

For each partition, the system call stack handles blocking and non-blocking system calls.

Statistics (such as high-water marks and margins) for system call stacks are persistent over partition restart.

For information on how to specify the size of a system call stack, see the *VxWorks 653 Configuration and Build Guide*.

Task Stacks

A task stack is the stack that is used for all routines that a particular task calls. Its size is defined when the core OS spawns the task. The kernel allocates the stack from the application domain's memory resources and adjusts the size as follows:

- Rounds up the size to a page boundary.
- If the core OS requested guard pages for stack overflow protection when it spawned the task, adds an additional mapped, but inaccessible page. For details, see the **taskLib** entry in the core OS.

Task Exception Stacks

Each task has an exception stack where the kernel saves system-critical information when the task encounters an exception. The **TASK_EXC_STACK_SIZE** configuration parameter determines the size of each task exception stack in the VxWorks 653 module. The kernel allocates the stack from kernel memory. The only user code that runs on this stack is user-supplied exception handlers; that is, routines that are connected to exceptions using **excConnect()**. During a system call, after the kernel saves system-critical data on the task exception stack, the kernel switches back to the task stack and the system call runs using that stack.

The kernel does not provide overflow protection on task exception stacks. The platform provider must either increase the size or ensure that handlers use only the minimal amounts.



WARNING: Increasing the size `TASK_EXC_STACK_SIZE` affects the entire VxWorks 653 module and can greatly increase memory usage.

Interrupt Stack

A VxWorks 653 module has one interrupt stack. It is similar to a task exception stack, except there is only one.

7.4 Shared Libraries

Shared libraries are used to share code among partitions. In a VxWorks 653 system, shared libraries can include only partition OS components and application code. Each VxWorks 653 module must contain at least one partition OS, and may contain one or more additional shared libraries. All shared libraries are created at system startup as shared library domains.

Any number of shared libraries may exist in a VxWorks 653 module. Shared library code can reference symbols declared as entry points in other shared libraries and in the partition's partition OS, but cannot reference entry points of the core OS.

VxWorks 653 shared libraries can attach to other shared libraries. They can also make calls to the partition's partition OS. However, circular dependencies are not permitted: no shared library should call another library that depends on it.

Partitions can attach to any number of shared libraries, as long as the exclusion restrictions specified by the included components are respected. Attachments are specified in the XML file for the partition and cannot be changed after system startup. A partition is attached only to the shared libraries specified directly in its **PartitionDescription** element. The attachment is not recursive. There is no limit for the length of the attachment chain, but it cannot be cyclical. For more information, see the *VxWorks 653 Configuration and Build Reference*.

Partitions attached to a shared library share their read-only sections (`.text`, `.rodata`), but have private copies of the writable data sections (`.data`, `.bss`, `.persistent.data`, and `.persistent.bss`).

7.4.1 Adding User-supplied Code to a Partition OS

You can add your own code to a partition OS. There are two ways to get its initialization routine (`myInit()` in the following example) to run:

- Add the following line to your code:

```
void * myInitFuncPtr _VTH_COM_INIT = myInit;
```

To avoid namespace conflicts, the `myInitFuncPtr` name must be either unique or defined as static.

The `_VTH_COM_INIT` macro (defined in `vxWorks.h`) causes `myInit()` to be called from `sslMain.c`, which is included with VxWorks 653.

If multiple initialization routines are defined this way, the routines are called in the order they are listed in the partition OS's dependency list.

or

- In the makefile, override `sslMain.c` and call `myInit()` there. For information, see the *VxWorks 653 Configuration and Build Guide*.

7.5 Shared Data Regions

A shared data region is a memory region or I/O region that can be accessed by one or more partitions. Shared data regions are implemented as shared data domains. They are created during system initialization according to the XML configuration file. For more information, see the *VxWorks 653 Configuration and Build Guide*.

A shared data region must have exactly one memory pool or one I/O pool associated with it. The pool is specified in the XML file for the shared data region. The physical and virtual address, when specified in the XML file, must be MMU page size aligned and must be valid addresses in the associated pool. When either the physical or virtual address is not specified (its value is `NULL`), they are allocated at run-time from the associated pool. The size of the shared data region

specified in the XML file must be a multiple of the MMU page size. For more information, see the *VxWorks 653 Configuration and Build Guide*.

In the core OS, shared data regions can be accessed using the **sdLib** API. You can get information about data regions with **pdShow()**. In the partition OS, the **sdRgnLib** library provides access to shared data regions attached by the partition.

Shared memory regions are always persistent: the kernel does not clear or store data when attached partitions are restarted. If the memory pool used to create the region is part of the system memory, the region is cleared during system startup. If necessary, it is the application's responsibility to preload any required data before the partitions are started.

Shared Data Region Configuration

The shared data region is defined in the XML file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*.

To get the information at run-time from the core OS, call **configRecordFieldGet()** with the **SD_RGN_CFG_RECORD** record and the appropriate selector as described below:

SD_RGN_MMU_ATTR

MMU attributes for the shared data region.

SD_RGN_NAME

Name of the shared data region.

SD_RGN_PHYS_ADRS

NULL or the physical map address. If the address is allocated at run-time from the associated pool, the field is NULL.

SD_RGN_POOL_NAME

Name of the shared data region's memory pool.

SD_RGN_SIZE

Number of bytes in the shared data region.

SD_RGN_VIRT_ADRS

NULL or the VxWorks 653 virtual address. If the address is allocated at run-time from the associated pool, the field is NULL.

Example 7-1 Accessing a Memory Region from a Partition

This example assumes that the VxWorks 653 module has a shared data region defined (in the XML file at configuration and build time) as follows:

```
<SharedDataDescription
  SystemAccess="READ_WRITE"
  UserAccess="READ_WRITE"
  DataType="DATABASE"
  CachePolicy="DEFAULT"
  Size="0">
  <Description
    Name="sdRgn1"
    Version="0.1.1">
```

This region has MMU settings that allow both reading and writing in the region. The virtual address is allocated at run-time from the associated memory pool, **sdRgn1Pool**. The partition running the following code must list this shared data region in its configuration in the XML file. For more information about configuring shared data regions, see the *VxWorks 653 Configuration and Build Guide*.

```
/******
 * sdRgnDemo.c - example showing usage shared data regions
 *
 * This example shows usage of a writable shared data region.
 * In this example, for simplicity, the same partition writes,
 * then reads data in the region.
 */

#include "vxWorks.h"
#include "sdRgnLib.h"
#include "stdio.h"

#define SD_RGN_NAME "sdRgn1"

typedef struct test_data
{
    int  data1;
    int  data2;
} TEST_DATA;

STATUS sdRgnDataSet
(
    char *      sdName,
    int         data1,
    int         data2
)
{
    void *      sdRgnAddr;
    TEST_DATA * pTest;

    /* get the base address of the shared data region */
```

```
    if ((sdRgnAddr = sdRgnAddrGet (sdName)) == (void *) NONE)
    {
        printf ("sdRgnAddrGet() failed for %s", sdName);
        return (ERROR);
    }

    /* set data in shared data region */

    pTest = (TEST_DATA *) sdRgnAddr;

    pTest->data1 = data1;
    pTest->data2 = data2;
    return (OK);
}

STATUS sdRgnDataShow
(
    char *      sdName
)
{
    void *      sdRgnAddr;
    TEST_DATA * pTest;

    /* get the base address of the data region */

    if ((sdRgnAddr = sdRgnAddrGet (sdName)) == (void *) NONE)
    {
        printf ("sdRgnAddrGet() failed for %s", sdName);
        return (ERROR);
    }

    /* print data stored in the region */

    pTest = (TEST_DATA *) sdRgnAddr;

    printf ("data in %s: %d %d", sdName, pTest->data1, pTest->data2);
    return (OK);
}

void sdRgnDemo ()
{
    if (sdRgnDataSet (SD_RGN_NAME, 123, 234) == ERROR)
        return;

    sdRgnDataShow (SD_RGN_NAME);
}
```

7.6 User Configuration Records

If memory for one or more user configuration record regions is configured in the core OS XML configuration file at configuration and build time, you can use that (read-only) memory. To access a particular user configuration record region, use the value of the **Base_Address** attribute of the appropriate **userConfigRecordRegion** element in the XML configuration file. For details, see the *VxWorks 653 Configuration and Build Reference*.

7

7.7 Multitasking

Multitasking in the core OS is similar to multitasking in a vThreads partition. For details, see [A.2 VxWorks Tasks](#), p.270 and [A.3 Intertask Communications](#), p.294.

7.8 Managing Memory

In a VxWorks 653 module, the core OS domain, shared libraries, partition OSs, and shared data regions each occupy a discrete space in virtual memory. However, application domains all occupy the same space in virtual memory. As a result, they are provided complete protection from errant code.

Each application domain has its own virtual memory context, consisting of a translation table (used to map virtual and physical memory) and other information about each page of memory. The task context of each task that holds a partition OS effectively includes the virtual memory context of the domain to which it belongs.

The core OS domain is mapped into the virtual memory context of each application domain, shared library, partition OS, and shared data region. However, a shared library, partition OS, or shared data region is mapped into the virtual context of an application domain only if the application attaches to it.

Applications can access kernel memory only by system calls to the kernel's API.

The kernel provides stack overflow detection: each task has a task exception stack, and the kernel maintains an interrupt stack for the entire VxWorks 653 module.

The kernel also reclaims resources to ensure that memory is freed whenever the owner of an object is deleted, thus helping to prevent memory leaks. (See [7.3 VxWorks 653 Stacks](#), p.141.)

In the core OS, the following types of memory management are available:

- **Managing Memory Partitions and Heaps**

Routines are available to manage memory partitions (including typed memory partitions) and the current heap. For details, see [7.8.1 Managing Memory Partitions and Heaps](#), p.148.

- **Managing Virtual Memory**

Routines are available to provide the following:

- caching on a per-page basis
- write-protection of text sections, read-only data sections, the exception vector table, and MMU translation tables

For details, see [7.8.2 Managing Virtual Memory](#), p.150.

- **Managing Page-oriented Memory**

With the routines that are available to manage page-oriented memory, the core OS can isolate and discretely manage each domain's memory. Routines are available to directly access each domain's virtual and physical pages. For details see, [7.8.3 Managing Page-oriented Memory](#), p.153.

7.8.1 Managing Memory Partitions and Heaps

Managing Memory Partitions

Memory partitions are contiguous areas of memory that the kernel uses to dynamically allocate memory. The **memPartLib** library lets the core OS do the following with memory partitions:

- Create and delete memory partitions by calling **memPartCreate()** and **memPartDestroy()**.
- Add memory to memory partitions by calling **memPartAddToPool()**.
- Allocate and free memory blocks from memory partitions by calling **memPartAlloc()**, **memPartAlignedAlloc()** and **memPartFree()**.

- Reallocate blocks of memory in memory partitions by calling **memPartRealloc()**.
- Set and get the options of memory partitions by calling **memPartOptionsSet()** and **memPartOptionsGet()**.
- Locate the largest free block in a memory partition by calling **memPartFindMax()**.
- Handle errors in memory partitions.

Managing Typed Memory Partitions

Kernel routines are available to create and access memory partitions that have specific memory-access permissions. Access permissions correspond to any valid combination of MMU attributes, including cache states (see [Table 7-1](#)). For example, the core OS can create a memory partition in which all allocated and free buffers are write-protected in supervisor and user modes. The **memAttrLib** library lets the core OS do the following with typed memory partitions:

- Create memory partitions with access permission attributes by calling **memAttrCreate()**.
- Allocate and free memory blocks from typed memory partitions by calling **memAttrAlloc()** and **memAttrFree()**.
- Copy data buffers to blocks allocated from typed memory partitions by calling **memAttrWrite()**.

Managing the Current Heap

The heap is the default memory partition from which the kernel dynamically allocates and frees blocks of memory. The kernel creates one heap for each application domain. The current heap is the heap of the current task's home domain. The **memLib** library lets the core OS do the following with the current heap:

- Allocate and deallocate memory blocks from the current heap by calling the ANSI-compatible **malloc()** and **free()**.
- Add memory to the current heap by calling **memAddToPool()**.
- Allocate memory aligned to a specific boundary by calling **memalign()** and aligned to a page by calling **valloc()**.

- Call the ANSI-compatible **realloc()**, **calloc()**, and **cfree()**.
- Locate of the largest free block in the current heap by calling **memFindMax()**.
- Set and get options for the current heap by calling **memOptionsSet()** and **memOptionsGet()**.

7.8.2 Managing Virtual Memory

Virtual memory contexts (also called virtual contexts) define the memory views that the core OS can access. The kernel creates a virtual context for each domain in the VxWorks 653 module. A virtual context includes virtual-to-physical page mappings, page access permissions, and page caching modes. The system virtual context is the virtual context of the current task's home domain.

The kernel provides an architecture-independent API of virtual-memory routines and the ability to do the following:

- Set the cache mode on a per-page basis.
- Write-protect text segments.
- Write-protect the VxWorks 653 exception vector table.
- Write-protect the virtual context translation table.

The kernel uses the MMU to create virtual-to-physical memory mappings. It also uses the MMU to enforce page-level attributes for access permissions and cache modes. Access permissions protect data and text from accidental corruption and prevent unauthorized (user mode) tasks from accessing supervisor text and supervisor data.

[Table 7-1](#) lists the MMU access and cache attributes that can be set on a page basis for page mappings on the PowerPC architecture.

The kernel sets the MMU's default caching policy based on the cache mode that the core OS specifies with **cacheLibInit()**. For example, if the core OS calls **cacheLibInit()** with **CACHE_COPYBACK**, the kernel sets **MMU_ATTR_CACHE_DEFAULT** to **MMU_ATTR_CACHE_COPYBACK**.

Table 7-1 MMU Page-Level Attributes

Description	Attribute
User mode read	MMU_ATTR_PROT_USR_READ
User mode write	MMU_ATTR_PROT_USR_WRITE
User mode execute	MMU_ATTR_PROT_USR_EXE
Supervisor mode read	MMU_ATTR_PROT_SUP_READ
Supervisor mode write	MMU_ATTR_PROT_SUP_WRITE
Supervisor mode execute	MMU_ATTR_PROT_SUP_EXE
Caching disabled	MMU_ATTR_CACHE_OFF
Copyback cache mode	MMU_ATTR_CACHE_COPYBACK
Write-through cache mode	MMU_ATTR_CACHE_WRITETHRU
Architecture-specific cache mode for I/O memory	MMU_ATTR_CACHE_IO
Default cache mode	MMU_ATTR_CACHE_DEFAULT
Architecture-specific MMU modes	MMU_ATTR_SPL_[0-7]

Accessing the MMU

The virtual memory library (**vmLib**) provides the core OS with an architecture-independent interface to the MMU. For details, see the reference entries for the core OS.

Ensuring Cache Coherency

Kernel facilities let the core OS perform DMA and interprocessor communication more efficiently by rendering associated buffers not cacheable. This is necessary to ensure that data is not buffered locally when other processors or DMA devices access the same memory location. Without the ability to make portions of memory

not cacheable, the core OS would need to turn off caching globally (resulting in performance degradation) or flush and invalidate buffers manually.

By calling **vmPgAttrSet()**, the core OS can change the MMU attributes of a block of virtual memory (a page). For example, pages can be defined as read-only or writable. Memory accesses to pages marked as not cacheable always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, hardware control registers, and systems without a bus-snooping mechanism.

Write-Protecting Text Segments

When a VxWorks 653 system is loaded, the kernel uses the MMU to prevent portions of memory from being overwritten. As a result, all text and read-only data are write-protected. Writing to write-protected memory causes a bus error.

For online-loaded partitions, the kernel marks text and read-only data sections as write-protected, so the core OS does not need to take additional steps to write-protect them.

The core OS can allocate and free memory blocks for the module sections using the routines in the **memAttrLib** library.

Write-Protecting the Exception Vector Table

During system initialization, the kernel write-protects the exception vector table. However, the core OS can change write-protection by calling **intConnect()**, which write-enables the table for the duration of the call.

Virtual Memory Contexts and Domains

The core OS domain is mapped into the virtual context of all applications, shared libraries, and shared data domains. The kernel pages are accessible only in supervisor mode.

Shared library domains to which an application is attached are also mapped into the virtual context of the application, but the pages corresponding to a shared library's writable data have a different mapping in each virtual context of the attached applications. These different mappings let all attached applications share a shared library's text and read-only data, but the writable data is private to each attached application.

The MMU mappings of the virtual pages that are available to each domain are maintained in the domain's virtual memory context. When the kernel schedules a task in another domain, the system's context is switched to the virtual memory context of the new task's domain.

7.8.3 Managing Page-oriented Memory

Managing Physical Memory

The kernel manages the physical memory that a domain can use as a set of pages in a physical page pool. The kernel uses the pool to allocate physical pages used for a mapping to virtual memory. When the core OS unmaps the pages, the kernel returns them to the pool. To ensure only one mapping for a given physical page, the kernel manages the allocation of all physical pages and allocates them only when the core OS requests a mapping. When multiple domains need to share the same physical page, a shared data region must be used.

The core OS manages page-oriented memory with routines in the **pgMgrLib**, **pgPoolLib**, and **pgPoolLstLib** libraries. The libraries use MMU hardware to map virtual and physical pages and to set access and cache modes for mappings made with routines in the **vmLib** library.

Configuring Physical Memory

For details on configuring physical memory (for example RAM, ROM, and I/O regions), see the *VxWorks 653 Configuration and Build Guide*.

Managing Virtual Pages

The kernel uses a virtual page pool to manage the virtual memory that it allocates to a domain. A domain has one virtual page pool associated with it. A domain also contains a physical page pool list that specifies the physical page pools that the domain can use.

When the core OS creates a domain, the kernel associates a page manager with the domain (the primary page manager). This page manager has access to all the physical page pools that are available to the domain. Therefore, the core OS can use the primary page manager's API to control any page that belongs to the domain.

In addition, the core OS can create additional, specialized page managers for a domain.

When the core OS allocates or maps a page, it can override the page manager's default MMU attributes and page-allocation policy.

The page manager libraries are **pgMgrLib** and **pgMgrShow**.

Creating Page Managers

The core OS can create a specialized page manager by calling **pgMgrCreate()**. This manager is in addition to the primary page manager that the kernel creates when a domain is created. The specialized page manager can be created to do the following:

- Allocate physical memory from a subset of the domain's physical page pools by specifying a physical page pool list parameter.
- Allocate mapped or unmapped virtual pages by setting the option to either of the following:
 - **PAGE_MGR_ATTR_ALLOC_MAPPED**
 - **PAGE_MGR_ATTR_ALLOC_UNMAPPED**
- Allocate contiguous or noncontiguous physical pages by default by setting the following options:
 - **PAGE_MGR_ATTR_ALLOC_CONTIG**
 - **PAGE_MGR_ATTR_ALLOC_NONCONTIG**
- Specify with options the default MMU attributes, protection and cache modes for mapped pages. (For information about page attributes, see [7.8.2 Managing Virtual Memory](#), p. 150.)

Getting a Page Manager's Current Options

The core OS can get a page manager's current options by calling **pgMgrOptsGet()**. In addition, the shell command **pgMgrShow()** displays information about a page manager.

Allocating Pages

The core OS allocates mapped or unmapped pages by calling **pgMgrPageAlloc()** or **pgMgrPageAllocAt()**. Both routines let you specify the following:

- Type of allocation; one of the following:
 - unmapped
 - mapped to contiguous physical memory

- mapped to noncontiguous physical memory
- MMU attributes (optional). If not specified, the page manager values are used.

In addition, **pgMgrPageAllocAt()** gives extra control over allocation by letting the caller optionally specify the virtual address, physical address, or both. If you call the routine with a specific physical address, the allocation type is mapped to contiguous physical memory, regardless of the options specified.

Mapping Pages

At any time, the core OS can map pages by calling **pgMgrPageMap()** and optionally specifying the MMU attributes and the type of mapping (to either contiguous physical memory or noncontiguous physical memory). If the MMU attributes or mapping are not specified, the page manager values are used.

Setting a Page's MMU Attributes

The core OS can set a page's MMU attributes by calling **pgMgrPageAttrSet()**. The result depends on the MMU architecture.

Getting a Page's MMU Attributes

The core OS can get a page's current MMU attributes by calling **pgMgrPageAttrGet()**. However, the returned attributes might not correspond to the attributes set with **pgMgrPageAttrSet()** or set when the page was allocated or mapped, because the page manager or MMU libraries might have changed the attributes to a set more appropriate for the architecture. For more information, see the **pgMgrLib** entry for the core OS and the appropriate MMU library reference.

Translating between Virtual and Physical Addresses

The core OS can translate between the virtual and physical addresses of mapped pages by calling **pgMgrVirtToPhys()** and **pgMgrPhysToVirt()**.

Unmapping Pages

The core OS can unmap pages by calling **pgMgrPageUnmap()**. The routine returns the page's associated physical pages to the appropriate physical page pools.

Freeing Pages

The core OS can free mapped or unmapped pages by calling **pgMgrPageFree()**. The routine returns the pages to the domain's virtual page pool, and they can then

be reallocated. The routine also unmaps all mapped virtual pages and returns the corresponding physical pages to the appropriate physical page pools.

Deleting Page Managers

The core OS cannot directly delete a domain's primary page manager. The core OS deletes a non-primary page manager by calling **pgMgrDelete()**. The routine removes only the control functionality. The status of all the pages that it allocated or mapped does not change. The core OS can later free or unmap the pages using the domain's primary page manager. The kernel reclaims all pages in the domain only when the core OS deletes the domain. At that time, the kernel unmaps the pages and releases the corresponding physical pages to the appropriate physical page pools.

7.8.4 POSIX Memory-Locking Interface

For details, see [5.3 POSIX Memory-Locking Interface](#), p.91.

7.9 Restart Functionality

VxWorks 653 supports the following four types of restart:

- **System cold start or restart**
This corresponds to the initialization or re-initialization of the whole VxWorks 653 module from power-on.
- **System warm restart**
This corresponds to the re-initialization of the VxWorks 653 module, following a power loss that lasts less than the time that causes RAM content to be lost.
- **Partition cold start or restart**
Initialization or re-initialization of a single partition. The RAM used by the partition is assumed to be corrupted.
- **Partition warm restart**
Re-initialization of a single partition.

7.9.1 System Cold Start or Restart

System cold start or restart has two versions, depending on which image is used.

- The ROM payload image loads from flash. This version is the one that is certified. It is described first.
- The RAM payload image is used during the development phase. It is described second, and is described by how it differs from the ROM payload version.

Deployed Configuration: ROM Payload Image

A system cold start of the ROM payload image consists of the following steps:

1. The boot loader code runs, initializing the CPU and RAM, then loading the core OS and the configuration record table from ROM to RAM and jumping to the core OS entry point.
2. The kernel initializes the hardware and runs any BSP initialization routines. It creates the kernel and initializes the partition scheduler.
3. The kernel parses the configuration records and creates partition OSs, shared libraries, and shared data domains according to their contents.
4. For each partition, including online-loaded partitions, the core OS performs the following operations according to the partition configuration information:
 - a. Creates the partition application domain.
 - b. Attaches the partition domain to all required shared libraries.
 - c. Attaches the partition domain to all required shared data domains.
 - d. Creates and initializes the partition health monitor context.
 - e. Creates the partition OS task. This is a user-level task that runs the partition OS as well as the partition application during the partition's schedule windows. This task is not activated at this point.
 - f. Maps the remaining RAM from the partition memory pool.
 - g. Initializes the VAL for this partition.
 - h. Creates the partition **restart** task. This is a core OS task that runs in supervisor mode and only during the partition's schedule window. The purpose of this task is to run the partition restart and shutdown operations. This task is activated, but runs only when partitioning is turned on.

5. The kernel notifies all partitions, except online-loaded partitions, to be cold restarted. This is achieved by requesting each partition **restart** task to run a cold restart on its dedicated partition. These partition cold restarts start only when partitioning is turned on.
6. Dynamic allocation in the kernel heap is disabled as well as dynamic mapping. This is done only if requested in the core OS configuration record (the **allocDisable** element in the XML configuration file). For more information, see the *VxWorks 653 Configuration and Build Reference*.
7. Enables partitioning and sets the first schedule-defined configuration record table. At this point, all the partitions start their cold restart during their own schedule windows, providing that a schedule window is defined in the initial schedule for the partition in question. Online-loaded partitions are not cold restarted. For the steps of a partition cold restart, see [7.9.3 Partition Cold Start or Restart](#), p.160.

For the steps of a system warm restart, see [7.9.2 System Warm Restart](#), p.159.



NOTE: The payload map is linked with the boot loader code at the end of the system build, resulting in file **sms_romPayload**. This file is programmed into the board flash or ROM. This lets the boot loader load both the core OS and the configuration record sections into RAM. Later, the kernel queries this payload map to load shared library and partition code. The payload map remains in flash or ROM.

Development Configuration: RAM Payload Image

In the development configuration, flash or ROM is replaced by RAM in order to accelerate the debugging cycle. The boot loader programmed in ROM or flash is used to download the single file of the RAM payload image into holding RAM. A specific region of RAM must be defined for this purpose in the XML configuration file. For details, see the configuration information in the *VxWorks 653 Configuration and Build Guide*.

The payload part of a RAM payload image consists of the core OS, shared libraries, partition OSs, configuration records, and applications. The remainder of the RAM payload image is the boot loader code linked with the payload map, resulting in the file **sms_ramPayload**. For more information, see the reference entry for **payloadLib** and the *VxWorks 653 Configuration and Build Guide*.

Once the boot loader loads the RAM payload image and jumps to the entry point of the image, the initialization steps are identical to the deployed configuration,

starting with the boot loader running. (In this configuration, the boot loader makes sure that only the operational RAM is initialized.)

7.9.2 System Warm Restart

In many VxWorks 653 modules, restarting the entire module can be desirable. The need to restart can have any number of reasons, but in almost all cases, user-supplied software is needed to accomplish the restart operation. This software needs to save the system state on a cold start, then restore it when a warm restart is required. It also needs to manage hardware-specific reset requirements before initiating the warm restart.

A large part of the time for a cold restart is spent loading RAM from the payload (typically, flash). The `coreOsWarmRestart()` routine is available to reduce this time and to do the following:

- Reset the processor state to the state that it was when the module was initialized.
- Reload any writable memory sections (such as `.data` and `.bss`). Persistent data and `.bss` data are untouched.
- Restart the core OS.



NOTE: The following pieces of code are assumed to be provided by the system integrator:

- power-down interrupt- and exception-handler code
- changes to `romInit.s` to reload the CPU context

Including Warm Restart in a BSP

To include warm restart in a BSP, a BSP developer must ensure the following:

- The BSP initializes the L1, L2, and L3 caches in `_sysInit()` (not in `sysHwInit()`).
- Level 3 cache routines initialize the following function pointers:
 - `_pSysL3CacheDisable`
 - `_pSysL3CacheEnable`
 - `_pSysL3CacheFlush`

- **_pSysL3CacheInvFunc**
- A initialization-stage function pointer (**_pSysHwRequiringMmuInit**) is initialized so that the BSP can initialize devices that require the MMU. If the function pointer is not **NULL**, its routine is called after the MMU is initialized.
- The BSP provides a routine (**sysHwCacheFuncsInit()**) to initialize the L2 cache function pointers and the above L3 cache function pointers.
- To use ROM payloads, the loader requires that the BSP map the ROM payload image (which may be in flash) to a BAT register. This mapping lets the shared library and partition sections be accessed properly from the shared library and partition contexts.

7.9.3 Partition Cold Start or Restart

For partition cold start or restart, the development system (the RAM payload image) behaves the same as the deployed system (the ROM payload image). The partition **restart** task is requested to run the cold restart operation of the specified partition. This request can be initiated by calling **partitionModeSet()** from code within the core OS. The health monitor may also call **partitionModeSet()**. Finally, the partition application can cold restart itself by calling **SET_PARTITION_MODE()**.

While initialization is in progress, preemption is disabled with lock level 0. As a result, process scheduling is disabled.

Once the partition **restart** task acknowledges the cold-start request, it performs the following main steps:

1. Flushes outstanding system calls.
2. Stops the partition OS task.
3. Zeroes the RAM assigned to the partition. For more details, see the reference entry for **partitionMemClearHookAdd()**.
4. Copies the partition text sections to RAM from ROM, flash, or RAM. In the case of a ROM payload image, the sections are copied from ROM or flash. A RAM payload image is copied from RAM.
5. Copies the partition **rodata** sections from ROM or RAM to RAM.
6. Copies the partition persistent and non-persistent data sections from ROM or RAM to RAM.

7. Zeroes the partition persistent and non-persistent **.bss** sections (this is skipped if step 3 was successful).
8. Copies the non-persistent and persistent data sections of shared libraries from ROM or RAM to RAM, for each shared library attached to the partition.
9. Zeroes the non-persistent and persistent **.bss** sections of shared libraries, for each shared library attached to the partition. This step is skipped if Step 3 was successful.
10. Resets the partition OS task so that it resumes running at the entry point of the partition OS. The application code starts running once the partition OS completes its initialization steps and all the shared libraries attached to the partition are initialized.

7.9.4 Partition Warm Restart

For partition warm restart, the development system (the RAM payload image) behaves the same as the deployed system (the ROM payload image). The partition **restart** task is requested to run the warm restart operation of the specified partition. This request can be initiated by calling **partitionModeSet()** from code within the core OS. The health monitor may also call **partitionModeSet()**. Finally, the partition application can warm restart itself by calling **SET_PARTITION_MODE()**.

While initialization is in progress, preemption is disabled with lock level 0. As a result, process scheduling is disabled.

Once the partition **restart** task acknowledges the warm restart request, it performs the following main steps:

1. Flushes outstanding system calls.
2. Stops the partition OS task.
3. Copies the partition non-persistent data section from ROM or RAM to RAM.
4. Zeroes the partition non-persistent **.bss** sections.
5. Copies the non-persistent data section of shared libraries from ROM or RAM to RAM, for each shared library attached to the partition.
6. Zeroes the non-persistent **.bss** section of shared libraries, for each shared library attached to the partition.
7. Resets the partition OS task so that it resumes running at the entry point of the partition OS. The application code starts running once the partition OS

completes its initialization steps and all the shared libraries attached to the partition are initialized.

The main difference between a partition cold and warm restart is that on a warm restart, the partition memory is assumed not to be corrupted. This is why only the data and **.bss** sections are re-initialized on a warm restart, while a partition cold restart clears the whole partition memory and reloads all the partition code. The other important difference is that persistent data variables are not re-initialized on a warm restart.

The application code can detect whether the partition re-initializes via a cold or warm restart by calling **GET_PARTITION_STATUS()** to retrieve the start condition information. For instance, an application may skip the re-initialization code of persistent data structures in the case of a warm partition restart. For more information refer to the reference entry for **GET_PARTITION_STATUS()**.

It is also possible to stop the execution of a single partition: the partition OS task is suspended after the pending system calls for that partition are flushed. The only way to have a partition run again after being shut down is to request a cold or warm restart on the partition. Again, **partitionModeSet()** or **SET_PARTITION_MODE()** can be used to request the shutdown of a single partition.

7.9.5 Restart Implications for Drivers

Core OS device drivers that are used by partitions should follow the following rules to make the VxWorks 653 module safe during partition restart.

1. The driver's **open()**, **creat()**, **remove()**, and **close()** should be deterministic in execution and bounded in time.
2. The **FIORESET** ioctl command code should be supported by the device driver. It is called during restart of a partition if it was in the midst of a **read()**, **write()**, or **ioctl()** operation on the device.
3. **FIORESET** should cause the thread of control, which is in the device driver doing a **read()**, **write()**, or **ioctl()** operation, to complete. **FIORESET** should *never* terminate the thread. Instead, the driver might do the following:
 - If the thread is blocking on I/O, wake the thread and cause it to return from the I/O operation.
 - If the thread is performing an I/O operation but not blocking, perform a **longjump()** of the thread so that it returns from the I/O operation.

7.9.6 Restart Implications for I/O

System warm restart and partition restarts (both cold and warm) are supported only if all the devices that include initialization routines and their corresponding I/O layers (for instance **ttLib** for SIO device drivers) are moved into the space that the partition can call directly. The one exception is timer drivers, which are left in the core OS.

A given device can be in either the core OS or a partition, but not both. When a device is moved into a partition, only that partition can control the device. In addition, the device can be configured in polling mode only. Interrupt mode is not supported when the device is moved into partition space.

A device configured to generate interrupts can reside in the core OS only.

For more information on vThreads I/O, see [9.2 I/O and vThreads](#), p.221. For more information COIL I/O see [9.4 I/O and COIL](#), p.264.

7.9.7 Persistent Data Support for Restart

Persistent data support is provided to let certain data in the partition OS preserve its value during partition warm restart. Such data is placed in several sections of the partition itself, or in a shared library that the partition attaches to. In the case of shared libraries, each partition has access to a private copy of the data, resident in the partition domain. It is this private copy of the data that can be modified by the partition.

When a partition runs a warm restart, all normal data sections except the **.persistent.data** sections are reloaded and all normal **.bss** sections are zeroed except the **.persistent.bss** sections. Persistent data sections (**.persistent.data**) are loaded only during a partition initial start or cold restart. Persistent **.bss** sections (**.persistent.bss**) are zeroed only during a partition initial start or cold restart.

Specifying Persistent Data

All persistent data is marked in the source code for placement into specifically named ELF sections. Initialized persistent variables are placed in a section named **.persistent.data**, while uninitialized persistent variables are placed into a section named **.persistent.bss**. In C code, this is achieved using the **__attribute__** GNU directive.

For instance, an initialized persistent variable is defined as:

```
int initializedPersist __attribute__((__section__(".persistent.data"))) = 123;
```

An uninitialized persistent variable is defined as:

```
int uninitializedPersist __attribute__((__section__(".persistent.bss")));
```

Unlike the normal uninitialized data in the **.bss** sections, the GNU compiler creates content for uninitialized persistent data in the **.persistent.bss** section. However, the **.persistent.bss** sections are excluded from the RAM and ROM payload images. Only their size and location are described in the payload map entries, as is the case for **.bss** sections.

How Persistent Data Is Handled

These elements always take up space in the **.sm** ELF module, unlike normal uninitialized data, which is placed into a **.bss** section. However, the **.persistent.bss** sections do not become part of a RAM or ROM payload image. Only their size and location are described in the payload map entries, as is the case for **.bss** sections.

When a partition runs a warm restart, all normal data sections (except the **.persistent.data** sections) are reloaded, and all normal **.bss** sections are zeroed (except the **.persistent.bss** sections). Persistent data sections (**.persistent.data**) are also reloaded during a partition cold start or restart. Persistent **.bss** sections, **.persistent.bss**, are zeroed only during a partition cold start or restart.

Important Limitation



CAUTION: It is important that the correct persistent section (**.persistent.data** or **.persistent.bss**) is selected in the source code. Neither the toolchain nor the kernel validates the selection.

For example, if a persistent initialized variable has the section attribute set to **.persistent.bss** in the source code, the kernel always initializes it to zero on cold restarts, even if its initial value in the ELF section was not zero. No check is performed to verify that only non-initialized persistent variables end up in **.persistent.bss** sections.

7.10 Partition Support

The API provided in the core OS can be used primarily to get configuration information and to control the operation of partitions. There are no direct API routines for creating or deleting partitions. Instead, partitions are created and initialized automatically by the kernel during its boot sequence according to the configuration information described in the configuration record data.

7.10.1 Core OS Partition-Related Components

INCLUDE_KERNEL_SHOW

This optional component enables **partitionShow()**.

INCLUDE_PARTITION_TOOL

This optional component provides support in the core OS for partition tools.

INCLUDE_WDB

This optional component allows collected activity data on sampling and queuing ports to be displayed.

7.10.2 Core OS Partition-Related Routines

The following core OS libraries relate to partitions:

- **configRecordLib**
- **partitionLib**
- **partitionShow**
- **payloadLib**

For details, see the reference entries in the *VxWorks 653 Core OS API Reference*.

7.10.3 Online-Loaded Partitions

This section provides example code for a loader that loads an online-loaded partition. It then changes the partition mode from the initial idle mode to cold-start mode so that the partition is scheduled to run. The code needs to be included in a kernel component. Additional code to call the loader needs to be added to a kernel component.

For information on configuring a system to use online-loaded partitions, see the *VxWorks 653 Configuration and Build Guide*.

Example 7-2 Online-Partition Loader: Example Code

```
/* Simple online-partition loader */

#include "vxWorks.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "taskLib.h"
#include "partitionLib.h"
#include "pdLib.h"
#include "pgMgrLib.h"
#include "private/pdLibP.h"
#include "vmLib.h"

#define BUF_SIZE          0x1000  /* 4 KB buffer */

STATUS onlinePartitionLoad
(
    char *fileName,
    int  partNum
)
{
    FILE *      file;
    UINT32      numBytes;
    PD_ID       kernelPdId = NULL;
    MMU_ATTR    mmuAttr;
    char *      payloadAddr = NULL;
    char *      pCurr;
    void *      address;
    REGION_NODE *pRgnNode;
    STATUS      status;

    if ((fileName == NULL) || (address == NULL))
        return (ERROR);

    kernelPdId = pdIdKernelGet();

    pRgnNode = rgnLookupByPoolName("onlinePayloadPool", 0);

    if (pRgnNode == NULL)
    {
        printf ("\nERROR - online partition payload region address lookup\n");
        return (ERROR);
    }

    address = (void *)pRgnNode->physAdrs;

    /* Find the virtual address of the online-partition payload */

    if (pgMgrPhysToVirt (kernelPdId->memInfo->primPgMgrId,
                        (PHYS_ADDR) address,
                        (VIRT_ADDR *) &payloadAddr) == ERROR)
    {

```



```

printf ("\nERROR - online partition payload address translation 0x%x (errno=0x%x)\n",
        (int)address, errno);

return (ERROR);
}

/* Write-enable the online-partition payload */

if ((pgMgrPageAttrGet (kernelPdId->memInfo->primPgMgrId,
                     (VIRT_ADDR) payloadAddr, &mmuAttr) == ERROR) ||
    (pgMgrPageAttrSet (kernelPdId->memInfo->primPgMgrId,
                     (VIRT_ADDR) payloadAddr,
                     pRgnNode->size / vmPageSizeGet(),
                     MMU_ATTR_SUP_DATA) == ERROR))
{
    printf ("\nERROR-couldn't write enable online partition payload source addr 0x%x \n",
            (int) payloadAddr);

    return (ERROR);
}

/* Load the online-partition payload */

pCurr = payloadAddr;

/* Open the file */

if((file = fopen(fileName, "r" )) == NULL)
{
    printf ("\nonlinePartitionLoad: failed to open file %s.\n", fileName);
    return (ERROR);
}

while (((numBytes = fread (buf, sizeof(char), BUF_SIZE, file)) != EOF)
        &&(numBytes != 0))
{
    /* Copy to the destination */

    bcopy (buf, pCurr, numBytes);
    pCurr += numBytes;
}

printf ("\nonlinePartitionLoad: load of %s at adress 0x%x successful.\n",
        fileName, (UINT) payloadAddr);

/* Reset the online-partition payload MMU attributes */

if (pgMgrPageAttrSet (kernelPdId->memInfo->primPgMgrId,
                     (VIRT_ADDR) payloadAddr, (pRgnNode->size / vmPageSizeGet()), mmuAttr) == ERROR)
{
    printf ("\nERROR - couldn't reset MMU attributes for online partition payload source
addr 0x%x \n", (int) payloadAddr);
    return (ERROR);
}

```

```
/* Change the partition mode from idle to cold start */

status = partitionModeSet (partNum, PARTITION_COLD_START,
    PARTITION_RESTART, "Cold Start of Online Loaded Partition", PART_ANY_PENDED);

if (status == ERROR)
{
    printf ("\nERROR - partitionModeSet failed, partition number %d errno 0x%x\n",
        partNum, errno);

    return(ERROR);
}
else
    return (OK);
}
```

7.11 Worker Tasks

Each partition can have worker tasks associated with it. They run in the context of the core OS to perform blocking operations (typically blocking I/O) on behalf of their partition. The core OS asynchronously passes back results to the partition. Worker tasks run within their partition's window (vThreads and COIL).

The number of worker tasks for a partition is specified by the **numWorkerTasks** element in the partition's XML configuration file. For details, see the *VxWorks 653 Configuration and Build Reference*.

To get the number of worker tasks at run-time, call **configRecordFieldGet()** with the partition configuration record and the **PARTITION_NUM_WORKER_TASKS** field selector.

If a partition is configured with no worker tasks, the core OS performs all system calls for the partition in the context of the partition OS. As a result, the entire partition is blocked until the call completes.

If an application makes so many system calls that it runs out of worker tasks, the partition blocks until a worker task is available.

If you include the vThreads target shell in a vThreads partition, two worker tasks need to be assigned to the partition shell task. The worker tasks are needed because the shell task pends on a read operation from the standard input file descriptor.

If no worker tasks are available to process a partition's blocking system call, the health monitor logs an event. This is an indication that more worker tasks might be needed.

The application developer or system integrator needs to understand the activity of the threads in the partition in order to determine whether they perform blocking operations in the core OS. The level of concurrency that the application requires also governs the number of worker tasks required.

The **valShow()** routine displays the value of the partition's **highWaterMark** field, which represents the maximum number of worker tasks that were dispatched at any time for the partition. You can use this value to determine whether the number of worker tasks needs to be adjusted.

7.12 System Time

All time values and capacities are unique and independent of partition execution.

System time is used for timestamping (**GET_TIME**) and is expressed in nanoseconds. Because kernel ticks do not provide nanosecond accuracy, an additional service provides a high-resolution time based on the current tick count and a hardware-based high-precision clock (decrementer, real-time clock, and so forth).

The kernel transmits clock ticks to each partition during its time window. Outside its own time window, a partition is not active and does not receive any clock ticks. At the next time window, the time of this partition is updated to reflect the absolute time of the VxWorks 653 module. The time update is done in a manner to ensure that no time event is lost in a delay queue.

7.13 Partition Scheduling

This section explains how the kernel schedules partitions. It also explains how to change schedules and scheduling.

By default, the kernel uses ARINC 653 scheduling to schedule partitions. ARINC 653 scheduling is time-preemptive scheduling (TPS). However, if there is idle time within a TPS schedule, the kernel uses priority-preemptive scheduling (PPS) for those partitions that have been enabled for it. The combination of the two scheduling methods is called APPS scheduling (ARINC plus PPS scheduling).

This section describes TPS scheduling first, then APPS.

7.13.1 TPS Scheduling

Scheduling Rules

TPS is strictly deterministic over time. The main characteristics are as follows:

- From the point of view of the core OS, the scheduling unit is a partition.
- Partitions have no priority.
- The scheduling algorithm is predetermined by the configuration file, is repetitive, and has a fixed periodicity.

At least one time period should be allocated to each partition during each cycle. There can be more than one assigned to a partition, and the partition windows need not be contiguous. It is acceptable to have some idle time within a time frame. During idle time, no partition is activated, and only kernel tasks can run. A partition is activated by allocating time to it within the major time frame. Each partition time period (or quantum) is defined by its offset from the start of the major time frame and its expected duration.

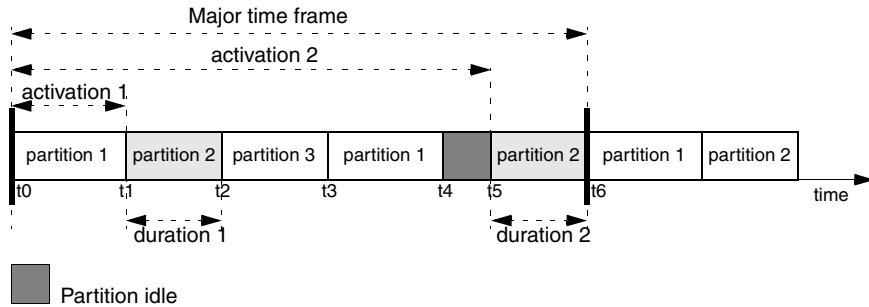
Core OS tasks associated with a partition are considered to be part of the partition itself and scheduled accordingly. For instance, the kernel tasks that process system calls on behalf of a partition are treated as part of the partition. How long a partition can run is defined in increments. The increment is configurable, but its minimum cannot be less than 0.25 milliseconds.

Partition Activation

The major frame is defined by its constituent quanta. These quanta consist of minor frames defined sequentially and specified by their duration and the partition to be scheduled. The activation time of a quantum is the sum of the preceding minor frame durations.

In Figure 7-2, both the partition attributes and the partition time frame table define the major time frame. The health monitor validates the content of the major time frame at startup time.

Figure 7-2 TPS Scheduling of Three Partitions



Partition attributes:

Quantum 1: Activation = t1
Duration = (t2 - t1)
Partition = 2

Quantum 2: Activation = t5
Duration = (t6 - t5)
Partition = 2

Spare-Time Monitoring

The kernel monitors spare time per partition and makes the information available for debugging. The information can be viewed using the host shell or target shell.

Mode-Based Scheduling

Mode-based scheduling is an enhancement to APEX that allows for a set of partition schedules to be defined and to be selectively enabled at the appropriate time by the kernel. It can support up to 16 partition schedules and a routine to allow transition between the schedules.

The routine supporting transitions between partition schedules is **arincSchedSet()** for the core OS and **SET_SCHEDULE_MODE()** for the partition. The routines let you select a transition at any of the following points:

- next major frame boundary
- next partition window boundary
- next timer tick

The transition jumps to the start of the new major frame.

The routines supporting transitions between partition schedules lets the caller be in either user mode or supervisor mode. (This lets schedules be changed by health-monitor fault-recovery routines, or by a privileged partition that acts as the mode manager for the VxWorks 653 module.) The routines supporting transitions between partition schedules are independent of other application APIs, so that access to them can be granted to specific partitions.

Multiple schedules can be configured for each partition by defining additional schedules in the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*.

7.13.2 APPS Scheduling

APPS scheduling allows for the VxWorks 653 module-wide scheduling of partitions in a global priority-preemptive scheme during a TPS schedule's idle time.

The kernel switches to PPS scheduling under either of these circumstances:

- There is idle time within a TPS schedule (that is, there is unused time left at the end of a running TPS partition window or the idle partition is scheduled next)
- The application forces idle time.

During PPS scheduling, all PPS-enabled partitions (and those configured with PPS priorities) are available for scheduling, and the non-idle partition with the highest PPS priority is scheduled to run. The partition configuration defines whether a partition is enabled for PPS scheduling. For details, see [7.2.1 Partition Configuration](#), p.135.

When a partition is scheduled to run during PPS scheduling, all threads within that partition run as they would normally.

TPS scheduling is not affected by PPS scheduling, which is strictly an alternate scheduling mechanism for partitions during idle time. If at any time during PPS scheduling, the TPS-scheduled partition that was idle must run due to an incoming non-tick pseudo-interrupt, PPS scheduling ceases, and the TPS-scheduled partition runs.

Pseudo-interrupts are the basic asynchronous communications mechanism from the kernel to the partition OS. They are used to deliver timer ticks, port message events, I/O subsystem events, and restart events.

While the scheduler is in PPS mode, if the TPS partition (which is idle) is delivered a pseudo-interrupt, PPS mode terminates, and the TPS partition starts to run immediately. This pseudo-interrupt is not a tick pseudo-interrupt (tick pseudo-interrupts are delivered to the current partition only), but rather a pseudo-interrupt generated by port message delivery, I/O subsystem responses, or warm restart.

Similarly, while the scheduler is in PPS mode, if a port message delivery, I/O subsystem response, or a warm restart is sent to an idle PPS partition, that partition becomes available for PPS scheduling, and may preempt the current (PPS) partition.

In addition, PPS scheduling is activated only if the amount of time remaining in a TPS partition window exceeds the value of an VxWorks 653 module-wide parameter (`PPS_ACTIVATION_WINDOW`, defined in `00comp_kernel_basic.cdf`).



NOTE: When designing partition schedules, only TPS time allocations should be taken into account, because PPS scheduling cannot provide any guarantee of CPU time.

When partitions are switched in either PPS or TPS scheduling, all partition switch hooks are run normally.

When PPS mode is switched back to TPS at the end of a TPS-scheduled window, if the same partition is to be scheduled to run (that is, the next TPS partition is identical to the current PPS partition), a partition switch is not performed.

Figure 7-3 shows TPS scheduling.

Figure 7-3 **TPS-Only Partition Scheduling**

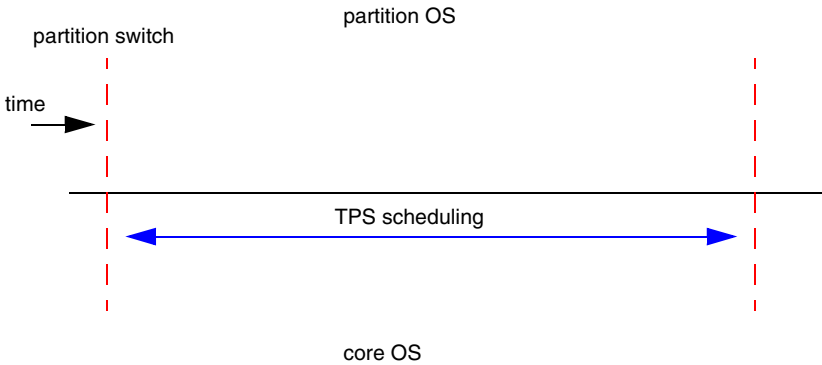


Figure 7-4 shows a PPS-enabled partition that runs without going idle. It is the same as for TPS-only scheduling.

Figure 7-4 **PPS-Enabled Partition Scheduling without Going Idle**

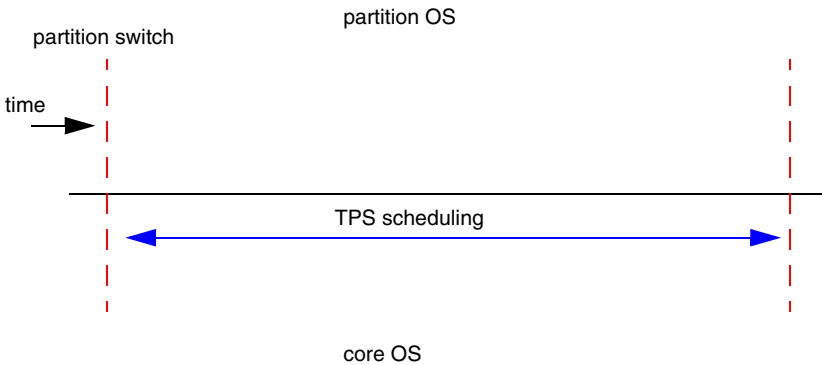
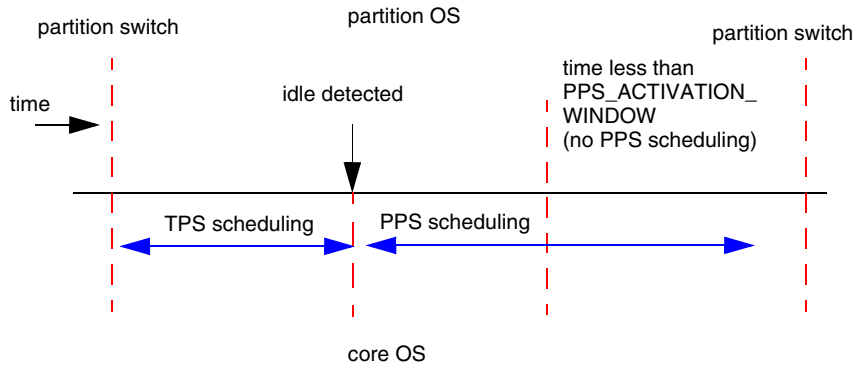


Figure 7-5 shows a PPS-enabled partition becoming idle and allowing PPS scheduling. Other partitions run during the PPS scheduling.

Figure 7-5 PPS-Enabled Partition Scheduling with Going Idle



How the Kernel Identifies Idle Time

Since ARINC scheduling is time-preemptive, platform providers often allocate generous amounts of time to partitions to ensure all their work can be performed. In pure TPS scheduling, after the work is completed, the partition remains the current partition, and simply waits until the next partition switch. In PPS scheduling, the partition OS (vThreads) determines this idle time by the following combination of factors, both of which must be true:

- There is a transition into the kernel idle state, and the amount of time remaining in the TPS partition exceeds the value of **PPS_ACTIVATION_WINDOW**.
- There is no delay operation in the partition (delayed task, deadline, or watchdog) whose delay could expire before the end of the current partition window.

When the partition OS in an ARINC partition determines that it is idle, there will be no threads ready to run.

Forcing Idle

Any partition can force the idle state by calling **appsIdleNotify()**. The routine indicates to the kernel that the partition has no ready-to-run threads and no

timeouts within its remaining partition window. (If timeouts do remain, they are ignored.) As a result, the only way for the partition to be rescheduled before its next TPS window is to receive a pseudo-interrupt.

vThreads and APPS Scheduling

When a vThreads partition is scheduled to run, vThreads schedules the highest-priority ready-to-run thread. vThreads continues to run and schedule threads until any one of the following occurs:

- vThreads goes idle.
- The application forces idle.
- A non-tick pseudo-interrupt is sent as a result of any API (port message, I/O subsystem, or warm restart). This could make another partition ready to run, and that partition could potentially be of a higher priority.
- The ARINC partition window expires.

If either one of the first two occurs, the partition is no longer available for PPS scheduling until such time as the following happens:

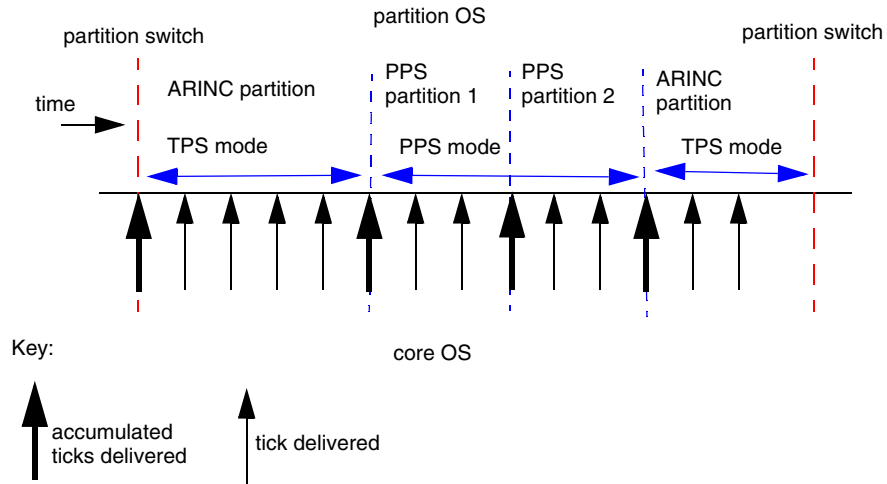
- Its next partition window.
- A non-tick pseudo-interrupt is sent to it.

Ticks and Timeouts

Ticks are delivered in PPS scheduling the same way as they are in TPS mode. This is true at the start of a partition window, whenever a PPS partition is scheduled, and if the TPS-scheduled partition again becomes ready to run due to a pseudo-interrupt (except a tick pseudo-interrupt.) For more information, see [Pseudo-Interrupts](#), p.177.

Figure 7-6 shows how timer ticks are delivered.

Figure 7-6 Delivery of Timer Ticks



Pseudo-Interrupts

In PPS scheduling, a non-timer pseudo-interrupt to a PPS-enabled partition causes the partition to be ready to run if it was previously not ready to run because of indicating an idle condition. This could cause a reschedule of which partition is the current highest priority.

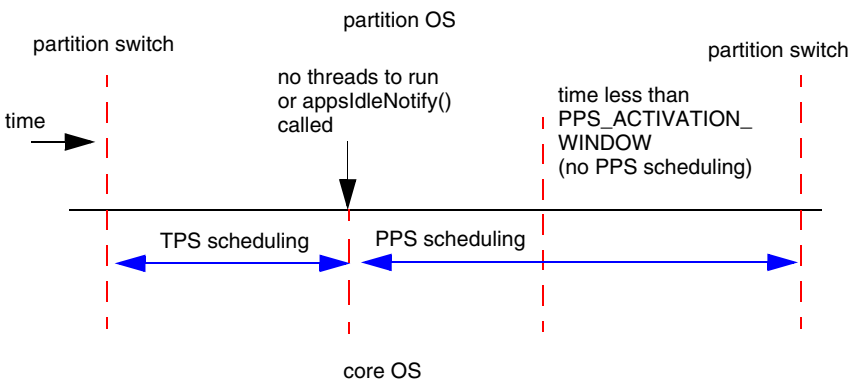
If a pseudo-interrupt is sent to the TPS-scheduled partition, the scheduling immediately switches from PPS back to TPS and lets the partition run until its window expires or the partition indicates another idle condition.

(Tick pseudo-interrupts are not delivered in PPS scheduling.)

Examples of APPS Scheduling

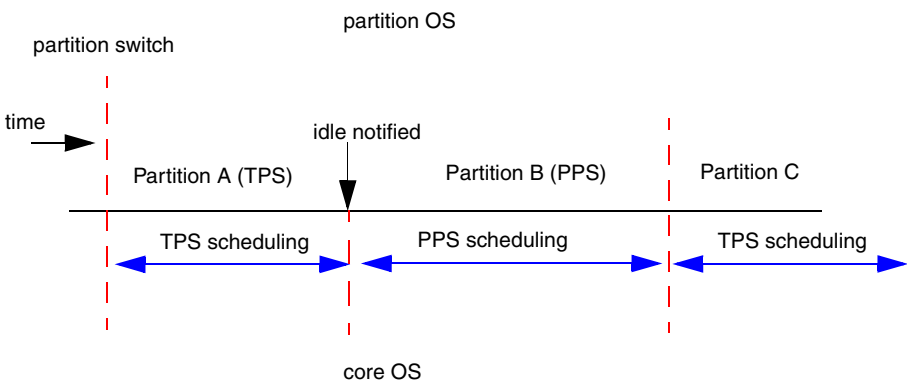
Figure 7-7 shows what happens when the TPS partition has no threads to run or calls `appsIdleNotify()`.

Figure 7-7 **TPS Partition with No Treads to Run or Calls `appsIdleNotify()`**



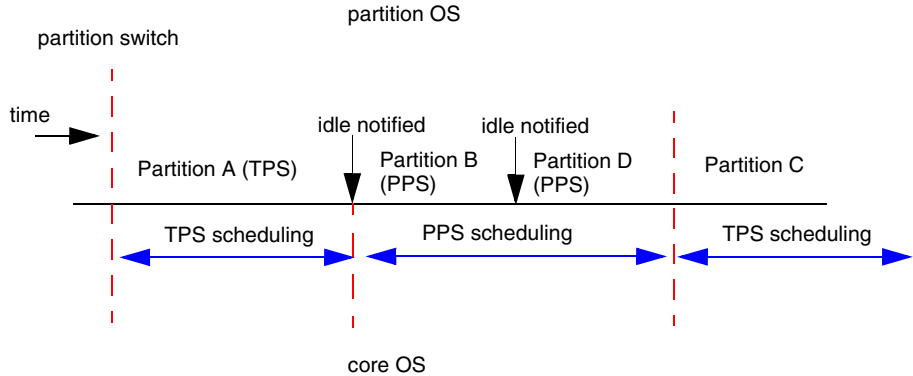
In Figure 7-8, Partition A (the TPS partition) runs and then notifies the kernel that it is idle. Partition B (the partition with the highest PPS priority) is subsequently allowed to run during PPS mode. At the start of the next TPS-scheduling cycle, Partition C is then run in TPS mode.

Figure 7-8 **TPS Partition Notifies Idle, PPS Partition Runs**



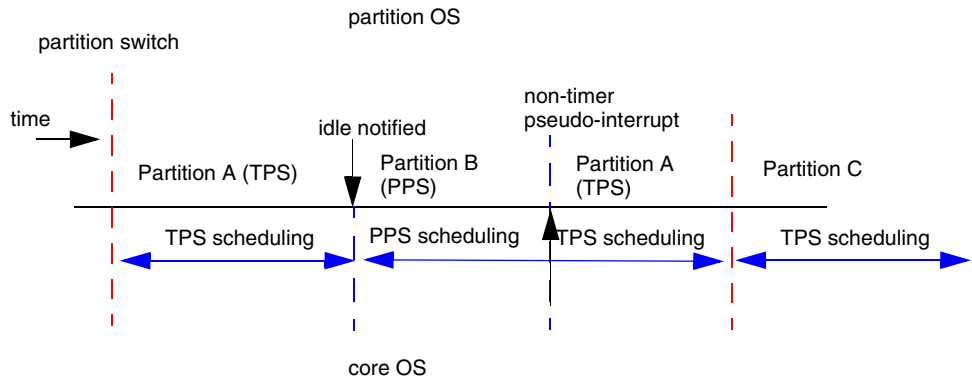
In Figure 7-9, Partition B (running in PPS mode) also goes idle. Partition D (the partition with the next-highest PPS priority) is then allowed to run.

Figure 7-9 TPS Partition Notifies Idle, PPS Partition Runs and Goes Idle, PPS Partition Runs



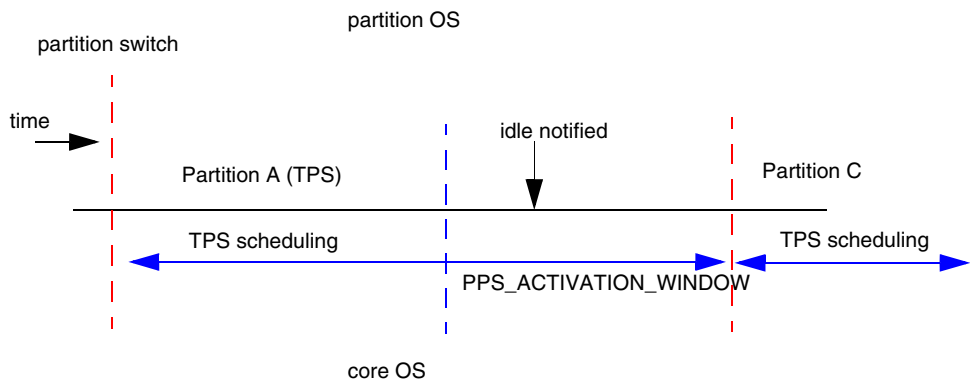
In Figure 7-10, while Partition B is running in PPS mode, a non-timer pseudo-interrupt (such as happens when a port call is made) is sent to Partition A (an ARINC partition). As a result, TPS scheduling is resumed and Partition A runs again.

Figure 7-10 PPS Partition Runs, Pseudo-Interrupt Occurs



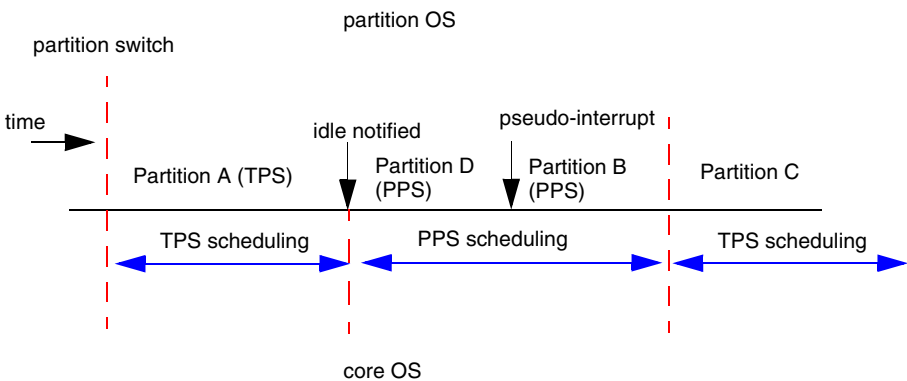
In [Figure 7-11](#), Partition A declares itself idle, but the idle declaration occurs within the `PPS_ACTIVATION_WINDOW` time at the end of the partition's time budget, and, therefore, PPS scheduling is not entered because there is not enough time.

Figure 7-11 **APPS Scheduling and PPS_ACTIVATION_WINDOW**



In [Figure 7-12](#), Partition D is running in PPS mode, and a pseudo-interrupt is sent to Partition B. Partition B has a higher PPS priority and is thus scheduled to run, preempting Partition D.

Figure 7-12 **PPS Partition Runs, Pseudo-Interrupt Preempts**



7.13.3 Partition-Scheduling Routines

The following partition-scheduling routines are available from the core OS:

- To register a callout routine that runs on partition context switches:
partitionSwitchHookAdd().
- To register a callout routine that runs on start of major frames:
partitionMajorFrameHookAdd().

7.14 Design Models for Ports

This section discusses the design models to support queuing and sampling ports in partitions.

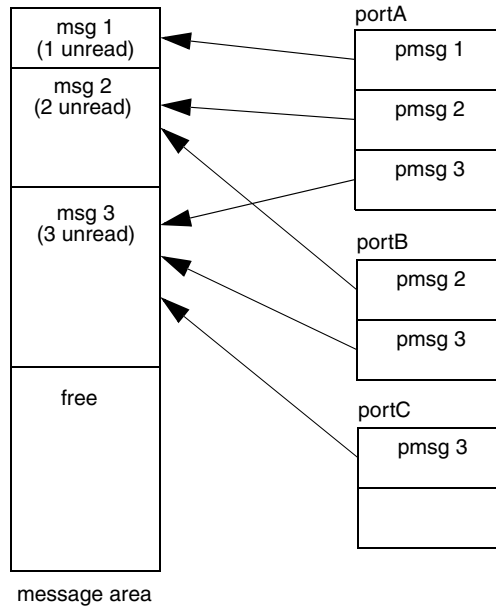
7.14.1 Design Model for Queuing Ports

The memory for queuing ports is in the kernel. The sending and receiving partitions access it through system calls, which validate all parameters. Messaging code resides in the kernel.

Memory Use

To reduce memory use, the kernel copies a message only once. Receiving partitions get the message from the same location. After the last destination port reads the message, the kernel makes the space available for new messages.

Figure 7-13 **Memory Model for Queuing Ports (Common Memory Used for All Queuing Ports)**



The source port is part of the message area. During transfers, the kernel does not move messages from source-port memory to destination-port memory. Instead, it moves only pointers to the messages. The maximum size of a message is the maximum size of the source port messages.

Messages and lists of pointers are in the kernel. They survive partition restart.

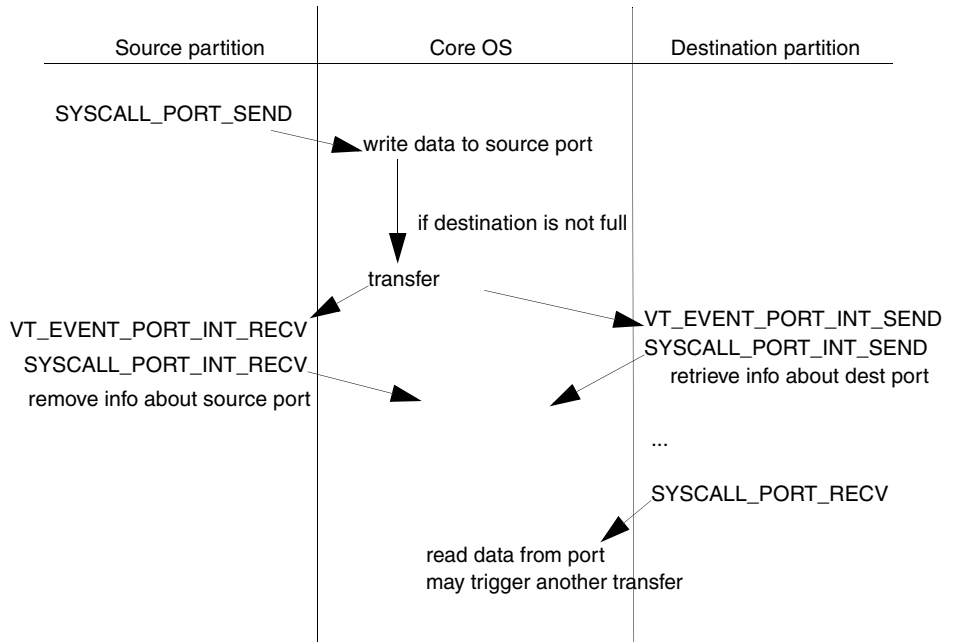
Blocking Processes

Since vThreads uses a many-to-many thread model (a worker-task mechanism) for system calls, sender processes that are blocked on full ports and receiver processes that are blocked on empty ports are queued in vThreads (user) space, not in kernel space.

System Calls and Events for Port Operations

Figure 7-14 shows the system calls and events that occur when a message is sent from a source port to a destination port.

Figure 7-14 System Calls and Events for a Port Operation



Effect of Restarting Partitions

When a partition is restarted, messages are lost only in the ports of the restarted partition. Therefore, when a source port's partition is restarted, its destination-port messages are preserved. Also, when a destination port's partition is restarted, its source-port messages are preserved.

7.14.2 Design Model for Sampling Ports

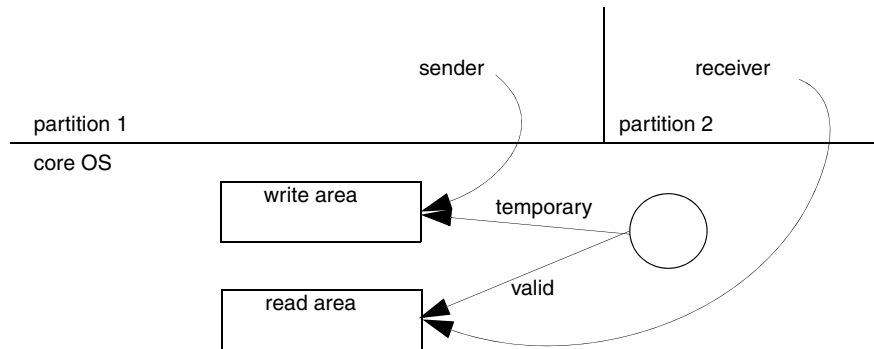
In sampling ports, messages carry similar, but updated, data. Messages and processes are not queued. A message remains in the source port until it is sent or

overwritten. Messages arrive in the order in which they are sent. When a new message reaches the destination port, it overwrites the previous message and remains there until it is overwritten itself. Sampling ports support variable-length messages.

The attributes of sampling ports are similar to those of queuing ports, but the behavior is different. The main difference is that a queuing port has a unique instance per message. As a result, there is no need to handle exclusion or synchronization. With sampling ports, there is only one message, and it can be overwritten at any time. The receiver could be reading data and get scheduled out. When it starts reading again at the next time window, it does not know whether a new message has replaced the original one.

Using a single data buffer to control access is not sufficient, because the sender or receiver would need to lock access during the write or read operation. If the sender or receiver got scheduled out, the data would remain locked, and no other partition could use it. Therefore, ports use a double buffer, as shown in [Figure 7-15](#). The sender uses a temporary buffer, and the receivers use a valid buffer. When the sender completes the write operation, it indicates changes in buffer status: valid becomes temporary, and temporary becomes valid.

Figure 7-15 **Design of Sampling Ports**



7.15 Setting up Communication with Other Modules

VxWorks 653 supports pseudo-ports and pseudo-partitions as defined by ARINC 653. As such, ports can be used to route messages to external modules or specialized devices, for example, to avionic busses.

In addition, partitions can communicate with other modules using partition direct-access ports. For details, see [4.8.2 Communicating Through Direct-Access Ports in a Partition](#), p.76.

Communication with external modules is made through pseudo-ports. A pseudo-port is a port attached to a pseudo-partition. It can be equated to routing (mapping) information required to direct messages between the source port and the destination port or ports over the intermodule communications channel.

When a channel's source port is configured with the **SENDER_BLOCK** message policy, only one of the channel's destination ports can be a pseudo-port.

Pseudo-ports are mapped to a particular supervisor-level driver, such as an AFDX device driver. For information on configuring pseudo-ports, see the *VxWorks 653 Configuration and Build Guide*.

A pseudo-port is identified by the following:

- a module-wide unique name
- whether it is direct access or not
- parameters (for example, size)
- its pseudo-partition
- the name of a supervisor-level driver

A direct-access pseudo-port is an APEX queuing port with no queuing. If queuing is needed, the driver must supply it. For information on how direct-access pseudo-ports might affect applications, see [4.8 Communicating with Other Modules](#), p.74.

The driver can service multiple pseudo-ports. It is identified by the following:

- a module-wide unique name

The APEX port library in the core OS uses the name of the driver to identify the set of routines to use when it deals with a pseudo-port. The name of the driver has no relationship to the name of the port driver.

- a set of routines

7.15.1 Configuring a Supervisor-Level Driver

The supervisor-level driver needs to be configured and added to the module, and this needs to be done before the APEX port library is initialized in the core OS. This configuration is achieved by adding the following to the core OS makefile. (In this example, **pseudoPortCreate()** is user-supplied code.)

```
PseudoPortComponent:
    prjCreate -type kernelComponent -build $(CPU)$(TOOL).debug -prjdir \
        $(PORT_DRIVER) \
        -srcfiles "$(SRC)/pseudoPortDrv.c"
    prj compAttributeSet -p $(PORT_DRIVER) CONFIGLETTES \
        "$(SRC)/pseudoPortCreate.c"
    prj compAttributeSet -p $(PORT_DRIVER) PROTOTYPE \
        "extern STATUS pseudoPortCreate (void);"
    prj compAttributeSet -p $(PORT_DRIVER) INIT_RTN "pseudoPortCreate();"
    prj compAttributeSet -p $(PORT_DRIVER) INIT_BEFORE "INCLUDE_APEX_PORT"
    prj compAttributeSet -p $(PORT_DRIVER) _INIT_ORDER "usrIosCoreInit"
    prj domComponentAdd -p $(BIN)/coreOS $(PORT_DRIVER) "INCLUDE_PORTDRIVER"
```

7.15.2 Adding a Driver

Based on the makefile in [7.15.1 Configuring a Supervisor-Level Driver](#), p.186, the configlet that initializes the driver calls the user-supplied **pseudoPortCreate()**. This routine needs to add the driver by calling **portPseudoDrvAdd()**.

The *portDrvName* argument is either the **DriverName** attribute in the XML port configuration or an appended value of it. For example, if the **DriverName** attribute is **/myPseudoDrvPort_net**, the *portDrvName* argument could be, for example, **/myPseudoDrvPort_net** or **/myPseudoDrvPort_net/1**.

If a device with the specified name already exists, the routine returns an error.

```
STATUS portPseudoDrvAdd
(
    PORT_DRV_FCT * pPortDrvFct, /* pointer to driver routines*/
    char * portDrvName, /* name of the driver */
    PORT_MODE_TYPE mode /* QUEUING or SAMPLING */
)
```

7.15.3 Driver Routines

The prototypes for the driver's routines are defined in:

installDir/target/h/apex/apexPortLib.h

The routines must follow the **PORT_DRV_FCT** definition, also defined in **apexPortLib.h**. (See [Function Pointer Structure for Drivers](#), p.189.)

Attaching the Name of a Driver to a Pseudo-Port ID

After the APEX port library initializes all ports and creates all channels, it calls the following routine to attach the name of a previously added (see [7.15.2 Adding a Driver](#), p.186) driver (*name*) to a pseudo-port ID (*pPseudoPortId*), which the routine creates and returns. The pseudo-port ID becomes an argument to the driver routines that get statuses, read, and write.

```
typedef STATUS (*PORT_Q_FUNCPTR_ATTACH)(
    char          * name,          /* name of the driver */
    PORT_ID       portId,         /* identifier created by the core OS */
    PORT_CFG_RECORD * pCfg,        /* XML configuration of the port */
    int           * pPseudoPortId, /* pseudo-port ID */
    PORT_MODE_TYPE mode          /* QUEUING or SAMPLING */
);
```

Reading Messages from a Pseudo-Port

The APEX port library calls the following routine to read messages from a pseudo-port.

```
typedef int (*PORT_Q_FUNCPTR_READ)(
    int           pPseudoPortId, /* pseudo-port ID */
    PORT_CFG_RECORD * pCfg,      /* XML configuration of the port */
    PORT_MODE_TYPE mode,        /* QUEUING or SAMPLING */
    PORT_MSG      * pOsMsg,     /* complete message structure */
    char *         pUserBuffer, /* partition payload pointer */
    SAP_ADDRESS_TYPE * pSrcUserHeader, /* SAP source header */
    SAP_ADDRESS_TYPE * pDstUserHeader, /* SAP destination header */
    UINT64         remainingTime, /* time left before switch out */
    BOOL           * canAcceptMore, /* another msg for next read? */
    int            * copyStatus,    /* status of the read */
    BOOL           * overflow,      /* TRUE returned if overflow occurs */
);
```

Service access point (SAP) ports are defined by the ARINC 664 specification, Part 7, which defines the Avionics Full Duplex Switched Ethernet (AFDX) protocol. SAP ports are used to communicate between AFDX systems and non-AFDX systems.

Writing Messages to a Pseudo-Port

The APEX port library calls the following routine to send a message to a pseudo-port.

```
typedef int (*PORT_Q_FUNCPTR_WRITE)(
    int                pPseudoPortId, /* pseudo-port ID */
    PORT_CFG_RECORD * pCfg, /* XML configuration of the port */
    PORT_MODE_TYPE    mode, /* QUEUING or SAMPLING */
    PORT_MSG          * pOsMsg, /* complete message structure */
    UINT64             remainingTime, /* time left before switch out */
    BOOL              * canAcceptMore, /* space available for next write? */
    char               * channelBuf, /* if not NULL, driver should use DMA to
                                     copy pOsMsg->payload data to channelBuf
                                     if possible */
    int                * copyStatus /* status of the write, return
                                     PORT_S_DMA_OPTIMIZED if channelBuf copy
                                     has happened */
);
```

Determining the Availability of a Pseudo-Port

The APEX port library calls the following routine to determine whether a pseudo-port is available to read from or write to:

```
typedef BOOL (*PORT_Q_FUNCPTR_AVAILABLE)(
    int                pPseudoPortId, /* pseudo-port ID */
    PORT_DIRECTION_TYPE direction /* SOURCE or DESTINATION */
);
```

For a pseudo-port to send or receive messages, the port must be available. The partition switch hook routine determines the availability by calling the driver's availability routine (**PORT_Q_FUNCPTR_AVAILABLE**). If the routine returns **TRUE**, the APEX port library can continue the read or write operation.

For a source pseudo-port, the APEX port library calls the driver's availability routine under the following conditions:

- The source pseudo-port was previously not available.
- The switched-in partition has a destination port connected to the same channel that contains this source pseudo-port.

For a destination pseudo-port, the APEX port library calls the driver's availability routine under the following conditions:

- The destination pseudo-port was previously not available.
- The switched-in partition has a source port or other destination ports connected to the same channel that contains this destination pseudo-port.

Getting the Status of a Pseudo-Port

The APEX port library calls the following routine to get the status of a pseudo-port. Status for a queuing pseudo-port is the number of messages stored at the port and the number of free messages.

```
typedef STATUS (*PORT_Q_FUNCPTR_STATUS) (
    int          pPseudoPortId, /* pseudo-port ID */
    PORT_INFO * pPortInfo
);
```

Determining Whether a Pseudo-Port Is Direct Access

The driver can determine whether a pseudo-port is direct access or not from the configuration record of the pseudo-partition to which the pseudo-port is attached. It can also determine the queue length from this configuration record. The driver can use the queue-length information to do its own queuing for direct-access pseudo-ports. Also, if the pseudo-port is direct access, the driver's read and write routines must act differently. For details, see [Sending Messages](#), p.189 and [Receiving Messages](#), p.190.

Function Pointer Structure for Drivers

```
typedef struct port_drv_fct /* function pointers for the driver */
{
    PORT_Q_FUNCPTR_ATTACH    attachRtn; /* called at port attach time */
    PORT_Q_FUNCPTR_READ      readRtn; /* read data from port */
    PORT_Q_FUNCPTR_WRITE      writeRtn; /* write data to port */
    PORT_Q_FUNCPTR_AVAILABLE availableRtn; /* get port availability */
    PORT_Q_FUNCPTR_STATUS     statusRtn; /* get port status */
} PORT_DRV_FCT;
```

7.15.4 Sending and Receiving Messages

Sending Messages

The APEX port library in the core OS uses the user-supplied write routine (*writeRtn()*) to write a message to a source queuing port. The routine has the same arguments as the `PORT_Q_FUNCPTR_WRITE` routine (see [Writing Messages to a Pseudo-Port](#), p.188) and returns the number of bytes written.

For direct-access pseudo-ports, if *writeRtn()* does not have time to write the message during the partition window, it must return **RETRY** to the kernel. The kernel propagates the status to vThreads, which in turn immediately retries the write operation.

The APEX port library creates a port-driver task and schedules it in the same window as the partition that owns the source port of the channel that contains the destination pseudo-port. The APEX port library activates the task each time a message cannot be distributed as a result of the APEX port library having called *writeRtn()*.

Receiving Messages

The APEX port library in the core OS uses the user-supplied read routine (*readRtn()*) to read a message from a destination queuing port. The routine has the same arguments as the **PORT_Q_FUNCPTR_READ** routine (see [Reading Messages from a Pseudo-Port](#), p.187) and returns the number of bytes read.

If the pseudo-port is direct access, *readRtn()* must ignore any specified timeouts and treat them as zero. Also, for direct-access pseudo-ports, if *readRtn()* does not have time to read the message during the partition window, it must return **RETRY** to the kernel. The kernel propagates the status to vThreads, which in turn immediately retries the read operation.

Time Partitioning

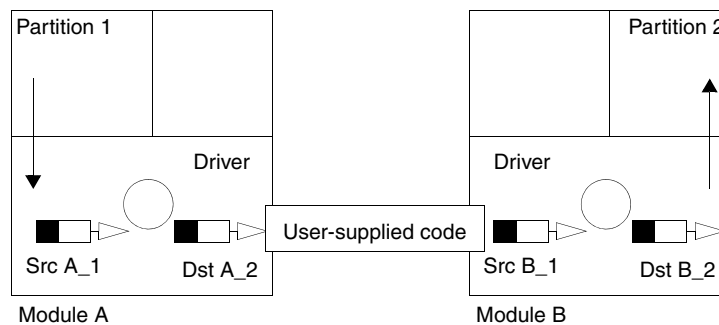
To avoid disturbing time partitioning with interrupts, the partition switch hook routine polls pseudo-ports to determine whether they are available to send or receive messages.

The *writeRtn()* and *readRtn()* routines have a *remainingTime* argument, which indicates to the driver the amount of time (in nanoseconds) until the current partition will be switched out. If the driver determines that it does not have enough time to copy the data, it should return **PORT_Q_RETRY** instead of **OK**, and the APEX port library retries the next time the partition is switched in.

7.15.5 Example: Communicating between Modules

In this example, a message is sent from one module to another. The pseudo-ports are not direct access. [Figure 7-16](#) shows the configuration.

Figure 7-16 **Example: Communication between Modules**



The source port on Module A (Src A_1) needs to connect with the destination port on Module B (Dst B_2).

The mechanism in this example allows for the one-to-many distribution of messages in both modules.

Configuration of Module A

The XML configuration for each module contains the channel definition. The configuration of Module A is as follows:

```
<PseudoPartition Name="pseudoPartitionA" Id="4" Type="IO_PARTITION">
  <PseudoPartitionDescription>
    <Ports>
      <QueuingPort
        Attribute="PSEUDO_PORT"
        Name="Dst_A_2"
        Direction="DESTINATION"
        MessageSize="500"
        QueueLength="100"
        DriverName="pseudoQPort"
        Protocol="NOT_APPLICABLE" />
    </Ports>
  </PseudoPartitionDescription>
```

```
</PseudoPartition>

<Applications>
  <Application Name="Partition 1">
    . . .
    <Ports>
      <QueuingPort
        MessageSize="500"
        Name="Src A_1"
        Direction="SOURCE"
        Protocol="SENDER_BLOCK"
        QueueLength="100"/>
    </Ports>
    </ApplicationDescription>
  </Application>
</Applications>
. . .
<Connections>
  <Channel Id="1">
    <Source PartitionNameRef="Partition 1"
      PortNameRef="Src A_1"/>
    <Destination PartitionNameRef="pseudoPartitionA"
      PortNameRef="Dst A_2"/>
  </Channel>
</Connections>
```

Configuration of Module B

The XML configuration for each module contains the channel definition. The configuration of Module B is as follows:

```
<PseudoPartition Name="pseudoPartitionB" Id="4" Type="IO_PARTITION">
  <PseudoPartitionDescription>
    <Ports>
      <QueuingPort
        Attribute="PSEUDO_PORT"
        Name="Src B_1"
        Direction="SOURCE"
        MessageSize="500"
        QueueLength="100"
        DriverName="pseudoQPort"
        Protocol="SENDER_BLOCK"/>
    </Ports>
  </PseudoPartitionDescription>
</PseudoPartition>

<Applications>
  <Application Name="Partition 2">
    . . .
    <Ports>
      <QueuingPort
        MessageSize="500"
        Name="Dst B_2"
```

```

        Direction="DESTINATION"
        Protocol="NOT_APPLICABLE"
        QueueLength="100" />
    </Ports>
</ApplicationDescription>
</Application>
</Applications>
.
.
.
<Connections>
    <Channel Id="1">
        <Source PartitionNameRef="pseudoPartitionB"
            PortNameRef="Src B_1" />
        <Destination PartitionNameRef="Partition 2"
            PortNameRef="Dst B_2" />
    </Channel>
</Connections>

```

User-Supplied Code for Module A's Send Operation

The user-supplied code in the core OS of Module A does the following for the send operation in this example:

- The user-supplied code registers the driver **pseudoQPort** by calling **portPseudoDrvAdd()**.
- (When the core OS APEX port library is initialized, it initializes all ports and also calls *attachRtn()* for Dst A_2.)
- The user-supplied code creates a port-driver task and associates it with the Partition 1 window. The task handles the distribution of Dst A_2 messages if needed.
- When the application in Partition 1 issues the **SEND_QUEUING_MESSAGE** service, the message is sent to Src A_1, and the user-supplied code marks this port as available.
- If Dst A_2 is available, the user-supplied code sends the message directly by calling the driver's *writeRtn()*.

If Dst A_2 is not available, the user-supplied code does not send the message.

- At each partition switch into Partition 1, the partition switch hook routine examines the state of Dst A_2 by calling the driver's *availableRtn()*. If the pseudo-port becomes available, the port-driver task is awakened to handle the distribution.
- After the port-driver task distributes the message, the user-supplied code removes the message from Src A_1 and calls the driver's *writeRtn()* to send the message to the external module.

User-Supplied Code for Module B's Receive Operation

The user-supplied code in the core OS of Module B does the following for the receive operation in this example:

- The user-supplied code registers the driver pseudoQPort by calling *portPseudoDrvAdd()*.
- (When the core OS creates the APEX port library, it initializes all ports and also calls *attachRtn()* for Src B_1.)
- The user-supplied code does not create a port-driver task, because the task is not needed for source pseudo-ports.
- When Partition 2 is scheduled and if Src B_1 is not available, the APEX port library calls the driver's *availableRtn()*. If Src B_1 becomes available, the state of Dst B_2 becomes available too.
- When the application in Partition 2 issues the **RECEIVE_QUEUING_MESSAGE** service, the APEX port library calls the driver's *readRtn()* for Src B_1 and distributes the message.

8

Health Monitoring

- 8.1 Introduction 195
- 8.2 Basic Health Monitor Concepts 196
- 8.3 Health Monitor Actions 210
- 8.4 Initializing the Health Monitor 215
- 8.5 Getting Health Monitor Information at Run-time 215
- 8.6 Defining the Health Monitor Handler Table 216
- 8.7 Health Monitoring for COIL Partitions 217
- 8.8 Other Facilities That Inject Alarms 218
- 8.9 Public Information 218

8.1 Introduction

Health monitoring provides a framework to raise and handle events, which can be alarms or messages, in a system. Alarms are injected to represent faults in the system, and handlers perform health recovery actions.

The bulk of this chapter describes health monitoring for vThreads partitions, and most of it also applies to partitions based on COIL. Differences are described in [*8.7 Health Monitoring for COIL Partitions*](#), p.217.

8.2 Basic Health Monitor Concepts

This section describes the following basic health monitor concepts:

- events (alarms and messages)
- health monitor hierarchy
- alarm injection
- thresholds

8.2.1 Health Monitor Events

An event is the base unit of injection within the health monitor. An event can be an alarm or a message.

Health Monitor Alarms

An alarm is an event that is the software representation of a fault that needs attention. It could have a positive or negative effect. Examples include hardware-generated exceptions, error paths in the code, and crossed thresholds.

Alarms have information associated with them. For details, see [Table 8-1](#).

Health Monitor Messages

A message is an event that has a code of **HM_MSG**. If the message is sent from within a partition, the partition health monitor handles it. If it is sent from outside the partition, the module health monitor handles it. A default handler is provided that logs the message. However, a system integrator can replace the handler with another by means of the XML configuration file. For more information, see the *VxWorks 653 Configuration and Build Reference*.

Messages, like alarms, have information associated with them. For details, see [Table 8-1](#).

8.2.2 Health Monitor Hierarchy

There are three levels of health monitoring:

- module health monitor
- partition health monitor
- process health monitor

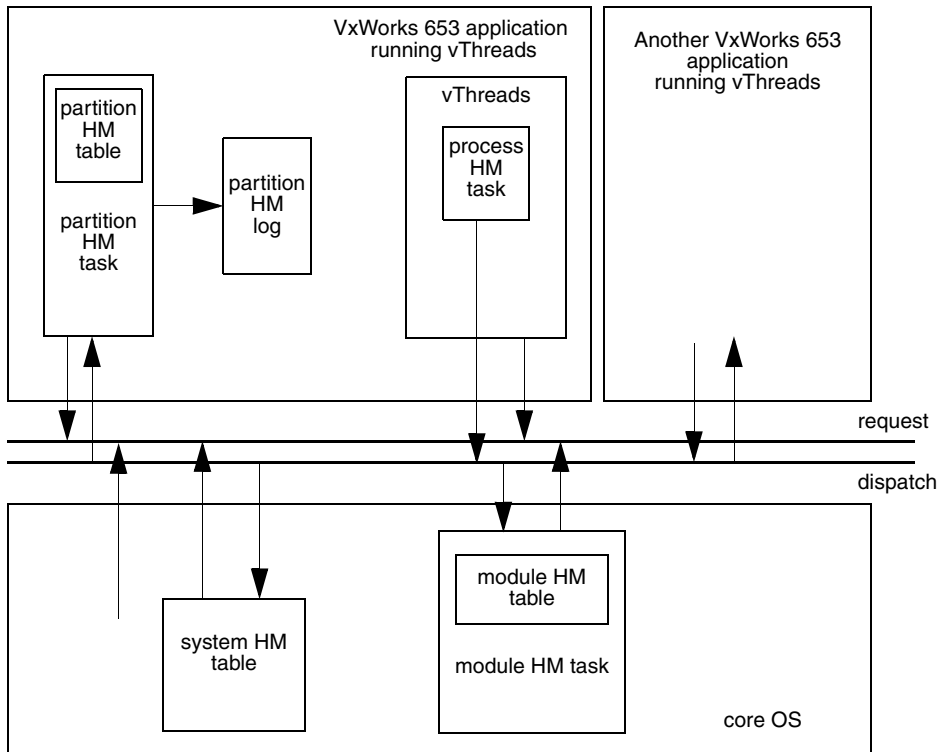
The partition health monitor and the module health monitor are driven by static tables that relate event codes to their appropriate handlers. There is one module health monitor table for the VxWorks 653 module. There is a partition health monitor table for each partition. The tables are loaded as part of the configuration loading for the VxWorks 653 module and partitions.

Messages have a hard-coded dispatch level. That is, the system health monitor table cannot be used to configure their dispatch level. At initialization, the hard-coded dispatch rules override anything in the XML configuration file that pertains to health monitor messages.

There is the potential for a process health monitor (also called the error handler process) for each partition. The application must create the process health monitor by calling **hmErrorHandlerCreate()** or, for ARINC 653 applications, by issuing **CREATE_ERROR_HANDLER**. The routines create a highest-priority task in the partition OS with which to run the process health monitor handler.

The relationship among the process health monitor, partition health monitor, and module health monitor, plus the general architecture of the health monitor from a scheduling perspective (not a memory containment perspective) is shown in [Figure 8-1](#).

Figure 8-1 Health Monitor Architecture (Showing vThreads Partitions)



After an event is injected, the system health monitor table determines how to dispatch the event. Because dispatching is hard-coded, events are automatically dispatched to the process, partition, or module level, depending on where they were injected.

The system health monitor table relates the event code and system status at injection time to a dispatch level, which can be one of the following:

- no level
- process level
- partition level
- module level

The system integrator creates the system health monitor table in the XML configuration file for the VxWorks 653 module. For information on specifying the table, see the *VxWorks 653 Configuration and Build Guide*. The table is read as part of the configuration tables of the core OS.

In terms of memory, the process health monitor (error handler process) is within the partition OS.

The partition health monitor runs as a core OS task (its stack is in the core OS kernel domain) with a higher priority than the core OS task that is running the associated partition OS (also higher than any worker tasks). But, it is scheduled during its associated partition's window.

The module health monitor runs as a highest-priority task in the core OS and is the only task at this priority. Its priority is higher than the partition health monitor tasks.

8.2.3 Event Structure (HM_EVENT)

Table 8-1 describes the fields of the structure (HM_EVENT) that defines a health monitor event.

Table 8-1 HM_EVENT Structure

Field	Description
code	The code associated with the event. If the code is HM_MSG, the event is a message.
subCode	The subcode associated with the event. If a subcode is not needed, the field is 0. For injecting events, applications must use HM_SUB_CODE_USER (defined in hmTypes.h) + offset. If the event has been reformatted, the field has the previous value of code. See Reformatting Events , p.208.
historicalCode	If the event has not been reformatted, the field is 0. If the event has been reformatted, the field has the previous value of subCode. See Reformatting Events , p.208.
level	The level at which the event was initially dispatched.
timeStamp	The time when the event was injected.

Table 8-1 **HM_EVENT Structure** (cont'd)

Field	Description
sysStatus	The status of the system when the event was injected. For details, see System Status and Modes , p.200.
addInfo	Additional information specified by the injector.
addr	The address where the injection was made.
partNumber	The partition number, indicating from which partition the event was injected. If the event was not injected from a partition, the field is 0.
taskName	A NULL-terminated string representing the name of the task that injected the event. If the event was injected from an interrupt context, the string is INTERRUPT .
taskId	The task ID of the task that injected the event. If the event was injected from an interrupt context, the field is 0.
msgLen	The length (in bytes) of the message body (msg). In the case of an exception, this is the sum of the sizes of " EXC_INFO\0 " and the EXC_INFO structure.
msg	<p>The message body, also known as the event payload and message payload. If the event is the result of an exception, msg contains text and data: the string "EXC_INFO\0" followed by the EXC_INFO data structure.</p> <p>If the event is the result of a second reformatting, see Reformatting Events, p.208.</p>

System Status and Modes

When an event is injected, the health monitor facility determines and sets the value of the **systemStatus** field in the **HM_EVENT** structure and passes it to the system health monitor, which uses it to determine the level to which to dispatch the event.

The **systemStatus** is a bitmap that can have one of the following values:

- **HM_MODULE_MODE**
- **HM_PARTITION_MODE**

- **HM_PROCESS_MODE**

Each injection of a health monitor event results in partitions or processes being preempted. The mechanism is not explicit. It results from the fact that the event is dispatched to a higher-priority task for handling, so that the partition OS scheduler preempts the current injecting task.

System Status for Core OS Context

For code running in the core OS context, system status can be the logical OR of the following:

- **HM_MODULE_HM_STATUS**
- **HM_MODULE_INIT_STATUS**
- **HM_PARTITION_HM_STATUS**
- **HM_PARTITION_SWITCH_STATUS**
- **HM_SYS_FUNC_STATUS**
- **HM_SYSCALL_STATUS**

8

System Status for Partition OS Context

For code running in the partition OS context, system status can be the logical OR of the following:

- **HM_PARTITION_INIT_STATUS**
(Even though this status corresponds to code running in the partition OS, the mode of the system is still partition mode, because the partition OS is not yet prepared to handle events.)
- **HM_PROCESS_EXEC_STATUS**
- **HM_PROCESS_MGMT_STATUS**

Module Mode (HM_MODULE_MODE)

Injections made from **HM_MODULE_MODE** are dispatched to the module level only. All partitions are preempted until the module-level handler handles the alarm. The handler can control the duration of preemption. But, since the situation that caused the alarm probably needs to be corrected before it is safe for partitions to continue running, the duration would not be an issue. If this is not the case, the alarm probably could and should have been handled by the partition health monitor.

HM_MODULE_MODE is equal to the logical OR of the following status values:

- **HM_MODULE_HM_STATUS**—module health monitor task
- **HM_MODULE_INIT_STATUS**—module initialization
- **HM_PARTITION_SWITCH_STATUS**—rescheduling as a result of a partition switch
- **HM_SYS_FUNC_STATUS**—not a health monitor task or a partition-related task; that is, an ISR
- **HM_UNKNOWN_STATUS**—unable to determine the system status.

Partition Mode (HM_PARTITION_MODE)

Injections made from **HM_PARTITION_MODE** are dispatched to the partition or module levels. The current partition is preempted, and the handler runs.

HM_PARTITION_MODE is equal to the logical OR of the following status values:

- **HM_PARTITION_HM_STATUS**—partition health monitor task
- **HM_PARTITION_INIT_STATUS**—partition OS initialization
- **HM_SYSCALL_STATUS**—in a core OS task that is related to a partition, but not the partition health monitor task

Process Mode (HM_PROCESS_MODE)

Injections made from **HM_PROCESS_MODE** are dispatched to the process, partition, or module levels. The current task in the partition is preempted until the alarm is handled.

HM_PROCESS_MODE is equal to the logical OR of the following status values:

- **HM_PROCESS_EXEC_STATUS**—running a partition OS task
- **HM_PROCESS_MGMT_STATUS**—in the partition OS kernel or partition OS interrupt state

8.2.4 Injecting Alarms

The core OS or an application injects an alarm by calling **hmEventInject()** or **HM_EVENT_INJECT()** with a code other than **HM_MSG**. ARINC 653 applications must issue the **RAISE_APPLICATION_ERROR** service with the **APPLICATION_ERROR** code.

Information about the alarm injection is collected from the viewpoint of where the injecting routine is called, which is not necessarily where the fault occurred. For instance, when **HM_EVENT_INJECT()** is called, the address is that of the program counter when **HM_EVENT_INJECT()** was called.

HM_EVENT_INJECT() is a macro to **hmEventInject()** that fills in the *addr* and *taskId* parameters to the program counter and the current task ID at the time that **HM_EVENT_INJECT()** is called.

The following routines can be used when calling **hmEventInject()**:

- **taskPcGet()**—Specifies the program counter of a non-running task.
- **vxCurrentPcGet()**—Specifies the current program counter.

The alarm goes first to the system health monitor table, which decides at what level to dispatch. [Figure 8-2](#) shows the logic of the alarm injection. [Figure 8-3](#) shows the subsequent dispatching to the appropriate level for handling.

Figure 8-2 Alarm Injection (vThreads)

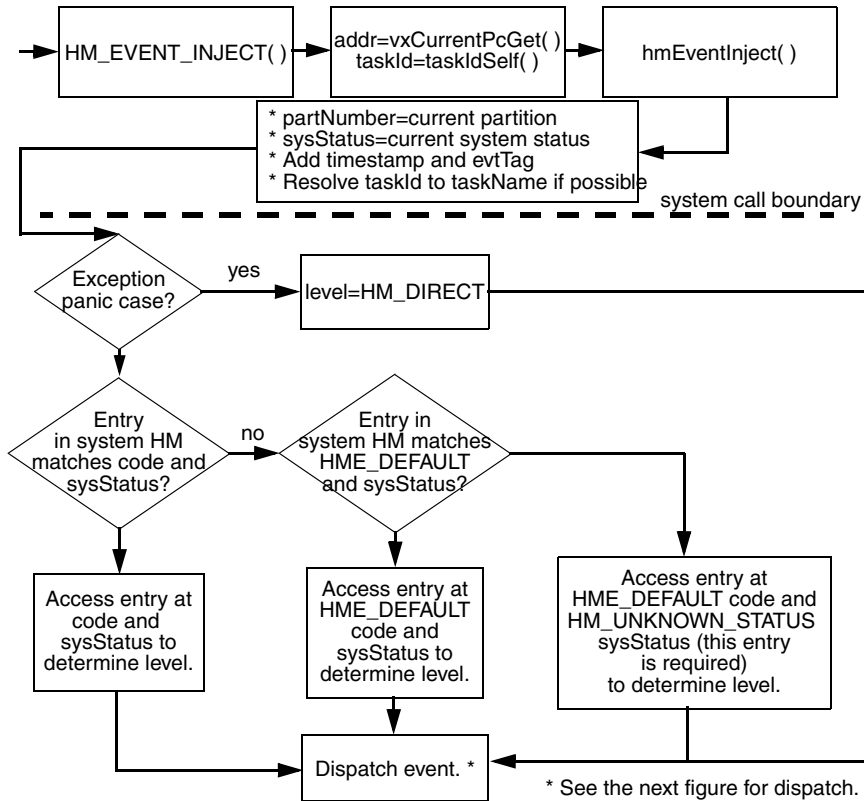
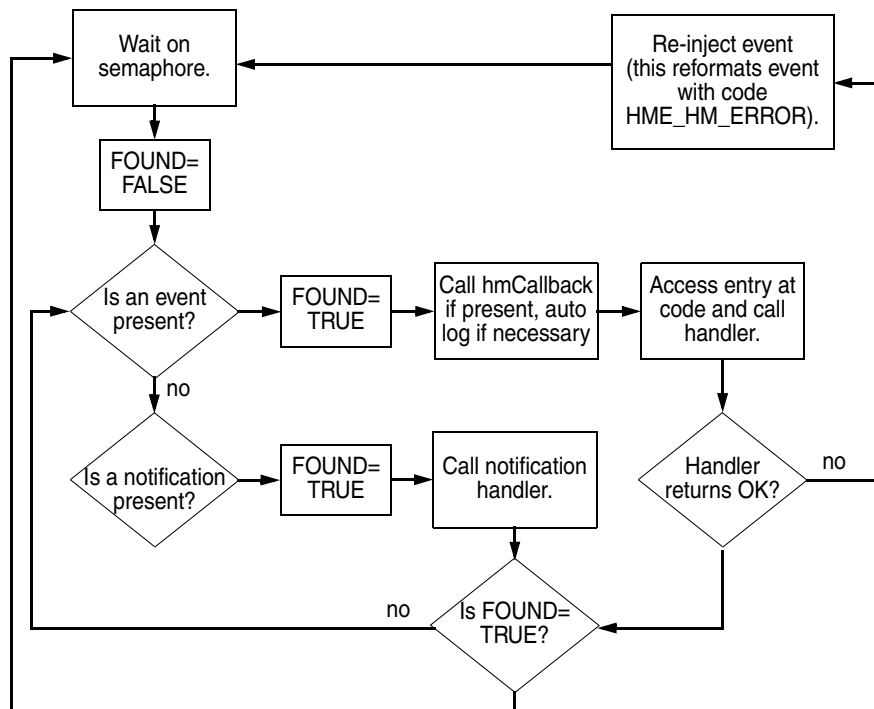


Figure 8-3 Alarm Dispatch (vThreads)



Dispatching Rules

This section outlines the rules for dispatching events. In the descriptions, interrupt context includes ISRs, watchdog routines, and kernel hooks (for example, partition switch hooks).

No Process Health Monitor Installed

Events dispatched to the process health monitor when the process health monitor is not installed are dispatched unchanged to the partition health monitor.

Alarms Injected in Exception or Interrupt Context

Alarms injected from exception or interrupt context are treated like alarms injected from task level. This means that the handling of events when injected from an ISR is deferred to task context. Except in the case of an exception panic, events injected due to an unhandled exception have their information (such as task information and system status) set according to what was happening at injection time.

In the case of an exception panic, the module-level handler is called directly. An exception panic can be any of the following:

- an exception in kernel state
- an exception from interrupt level
- a nested exception
- an exception in the root task

Events Injected from Tasks in the Partition OS

Events injected from a task in the partition OS that has a priority equal to the process health monitor (error handler) task, which ordinarily are dispatched to the process level, are dispatched unchanged to the partition level. This includes injection from the process health monitor task. The exception is for the **APPLICATION_ERROR** event code, which can be injected from the error handler process and handled by it.

Events Injected from Tasks outside the Partition OS

- **Priority Greater than or Equal to the Partition Health Monitor Task**

Events injected from a task outside the partition OS that has a priority equal to or greater than the partition health monitor task, which ordinarily are dispatched to the partition level, are dispatched unchanged to the module level. This includes injection from the partition health monitor task.

- **Priority Equal to the Module Health Monitor Task**

Events injected from a task outside the partition OS that has a priority equal to the module health monitor task, have their handlers called directly and synchronously. This applies to injection from within a module health monitor handler only.

Full Health Monitor Queues

If dispatch is not possible because the partition health monitor queue is full, the event is reformatted with the **HME_HM_ERROR** code and dispatched to the partition according to the rules for injecting from the partition health monitor task.

If dispatch is not possible because the process health monitor queue is full, the event is reformatted with the **HME_HM_ERROR** code and dispatched to the partition according to the rules for injecting from the process health monitor task.

If dispatch is not possible because the module health monitor queue is full, the module-level **HME_HM_ERROR** handler is called directly and synchronously according to the rules for injecting from the module health monitor task.

Task-Lock Condition Exists

If an event is injected while a task-lock condition exists, the following rules are followed:

- **Injected from the Task Context**

If an event is injected from the task context while a task-lock condition exists, the lock is broken and the error handler process runs. However, before breaking the lock, the partition OS raises the task's priority to one less than the error handler's priority. This strategy is an attempt to have the task regain its task lock. The preempted task then runs immediately after the error handler, at which point the partition OS restores the task-lock count and original priority. However, if an application uses task priorities 0 or 1 (contrary to the ARINC 653 specification), there is no guarantee that the preempted task is the first to run after the error handler.

- **Injected from the Interrupt Context**

If an event is injected from the interrupt context while a task-lock condition exists and **watchDogDuration** (as specified in the partition XML configuration file) is 0, locks (preemption and task locks) are broken immediately and the error handler process runs. The error handler process restores the locks and instructs the partition OS to run the preempted task first (unless the task has stopped).

If **watchDogDuration** is **INFINITE_TIME**, the error handler process runs when the task unlocks itself.

If **watchDogDuration** is between 0 and **INFINITE_TIME**, any APEX locks (the result of the application issuing the **LOCK_PREEMPTION** service) are broken immediately. If there are any other task locks remaining (the result of the kernel calling **taskLock()**), the watchdog is started. When the watchdog expires, task locks are broken, and the error handler process runs. The error handler process restores the locks and instructs the partition OS to run the preempted task first (unless the task has stopped).

Handlers Cannot Handle the Alarm

Handlers that cannot handle an alarm must inject an alarm of their own or return **ERROR** so that the health monitor can reformat the event with the **HME_HM_ERROR** code.

However, returning **ERROR** does not apply at the process health monitor level. At this level, the application must inject another event from the process health monitor, which the health monitor reformats with the **HME_HM_ERROR** code. For rules pertaining to injecting from the process health monitor task, see [Events Injected from Tasks outside the Partition OS](#), p.206.

All system health monitor tables should include a handler for **HME_HM_ERROR**.

The return code from calling an **HM_DIRECT**-level handler is not checked, because no further escalation is possible. If an **HME_HM_ERROR** module-level handler is not declared (or any handler for that matter), the **HME_DEFAULT** handler is called.

Reformatting Events

When the health monitor facility reformats an event, the event gets a new **code**. The old **code** moves to the **subCode**, and the old **subCode** moves to the **historicalCode**. If this is a second reformatting, **historicalCode** information would be lost. In this case, an event is injected with the following information:

- **code** of **HME_DATA_LOSS** (**subCode** of 0, **historicalCode** of 0)
- **msg** containing the old **code**, **subCode**, and **historicalCode** (retrievable by casting **msg** as an **HM_DATA_LOSS** data structure)

Dismissing Alarms

An alarm is dismissed under any of the following circumstances:

- In the partition health monitor table or module health monitor table, for a given **code**, a NULL handler (**CFG_NO_HANDLER**) is set.
- In the system health monitor table, for a given **code** and **systemStatus** combination, the dispatch level is set to **HM_NO_LVL**.

Dispatching and Logging Messages

A message event differs from an alarm event as follows:

- Alarms are dispatched according to the system health monitor table. Messages are not dispatched by this mechanism. If a message is injected within the partition, it is dispatched to the partition health monitor. If it is injected outside the partition, it is dispatched to the module health monitor. Each level has a default handler that logs the message to the partition or module log. The system integrator can replace the default handler through the XML

configuration file. For more information, see the *VxWorks 653 Configuration and Build Reference*.

- Alarms are logged only if automatic logging is enabled or if a handler explicitly logs the alarm by calling **hmEventLog()**. The system integrator must specify the handler in the XML configuration file. For information, see the *VxWorks 653 Configuration and Build Guide*.

8.2.5 Health Monitor Thresholds

Thresholds apply to the following:

- notification queues
- logs
- event queues

8

Notification Queue Threshold

The notification queue threshold equals the depth of the notification queue. An event is injected when the notification queue overflows. For information on enabling and disabling overflow notification, see the *VxWorks 653 Configuration and Build Reference*. If overflow notification is disabled and if an event is injected because of overflow, the event is dispatched to the partition health monitor or module health monitor with which the notification queue is associated.

Log Threshold

The log threshold defines if and when an event should be injected when the log has a certain number of entries. For information on specifying, enabling, and disabling the log threshold, see the *VxWorks 653 Configuration and Build Reference*.

Event Queue Threshold

The event queue threshold defines if and when an event should be injected when the event queue has a certain number of entries. For information on specifying, enabling, and disabling the event queue threshold, see the *VxWorks 653 Configuration and Build Reference*.

Error Handler Queue Threshold

The error handler queue threshold defines the event threshold of the error handler. It is analogous to the event queue threshold that applies to the partition health monitor or module health monitor.

8.3 Health Monitor Actions

The health monitor facility does the following when an application runs:

- escalates alarms
- logs events
- notifies other partitions
- issues a callback

In addition to detecting and reporting its own faults, an application needs to respond to the above actions.

8.3.1 Escalating Alarms

Alarms are not automatically escalated, because it is the system integrator who knows the level and handler that best services each alarm. The system integrator configures the system health monitor table in the XML configuration file. For details, see the *VxWorks 653 Configuration and Build Guide*.

If the specified handler from the partition health monitor table or module health monitor table cannot handle the alarm, it should return **ERROR** or inject an alarm of its own.

If a handler returns **ERROR**, this indicates that the alarm was not handled correctly. In this situation, the health monitor facility reformats the alarm, using the first alarm information, but with **code** equal to **HME_HM_ERROR** (see [Reformatting Events](#), p.208).

As a result of the above, the system integrator can choose whether to handle alarms that result from alarms not being handled and, if so, which ones.



NOTE: Only the health monitor facility can inject alarms with the **HME_HM_ERROR** code. This is enforced so that a rogue task cannot directly inject these types of alarms, possibly forcing escalation and affecting the protection guarantees between partitions.

8.3.2 Logging Events

For information on configuring, enabling, and disabling the logging of alarms and messages, see the *VxWorks 653 Configuration and Build Reference*.

When logging is enabled, if an event is injected from within a partition (**sysStatus** is **HM_PROCESS_MODE** or **HM_PARTITION_MODE**), the event is logged to the partition health monitor log.

When logging is enabled, if an event is injected from outside the partition (**sysStatus** is **HM_MODULE_MODE**), the event is logged to the module health monitor log.

When logging is not enabled, a handler must call **hmEventLog()**, which might generate an additional system call to log the alarm, depending on where the current event processing is occurring.

Application code can log messages to the health monitor log by calling **hmEventInject()** with the **HM_MSG** code (assuming the default handler behavior).

There is one log for each partition and one log for the VxWorks 653 module. By default, the logs are stored in volatile memory. If logs need to survive module or system restart, handlers need to be provided that write the logs to non-volatile memory.

Each log is a circular buffer of configurable size. As a result, for log sizes greater than zero, the request to log an event is never denied. For a log size of *n*, only the *n* most-recent entries are in the log. Old information could be overwritten.

Logs can be accessed by calling **hmLogEntriesGet()** with the partition ID and the number of entries to retrieve. Partition ID 0 accesses the module log. The log can be read in FIFO or LIFO order. Entries that have been read can be preserved in the log or purged. An offset can be specified to retrieve the log in segments. Read-purge with LIFO-order reading is not permitted, because it would result in a discontinuous log. [Table 8-1](#) lists the information in each entry of each log.

8.3.3 Notifying Other Partitions

When an alarm is injected and fault-recovery action is taken, other partitions might want to be notified. For example, partitions might want to know when another partition shuts down. All calls to notify and to notification handlers reside in the core OS. As such, notification does not apply to the error handler process.

A partition must register in order to be notified of events. The system integrator defines this in the XML configuration file by specifying, for the VxWorks 653 module and each partition, to which event code it wants to be notified and from which partitions it will accept notification (known as trusted partitions). This combination of event codes and trusted partitions is called an allowable notification. For configuration details, see the *VxWorks 653 Configuration and Build Reference*.

The notification facility creates a queue for each partition that wants notification. The health monitor task (either the partition health monitor task or module health monitor task) services its notification queue and services it after servicing its event queue.

A module or partition health monitor handler notifies partitions of an event by calling **hmNotificationSend()**. The partition sending the notification is determined by where the call to **hmNotificationSend()** is made and not by where the event was injected. If the call to **hmNotificationSend()** is made from outside the context of a partition health monitor, the notification is considered to be from the VxWorks 653 module and, hence, trusted by all.

Each partition has a registered notification handler that is called in response to an allowable notification. The associated queue holds messages of **HM_EVENT** type. If the queue fills, the notification agent does not block. Instead, it injects an event to the associated partition health monitor or module health monitor and flags the queue as invalid. This action excludes that partition from future notifications. The **code** of this event is **HME_HMQ_OVERFLOW** and the **subCode** is **HME_HMQ_OVERFLOW_NOTIF**. The event handler must fix the problem and then re-register the partition with the notification agent by calling **hmNotificationReReg()**. The queue is not flushed until the partition re-registers for notification. This gives the handler for **HME_HMQ_OVERFLOW** a chance to flush the queue and preserve data before the queue is forcefully flushed.

8.3.4 Issuing Callbacks

The health monitor callback facility can be used for such things as error reporting to an external entity or NVM file system. It is enabled by specifying a non-NULL

name for the health monitor callback routine in the XML configuration file. This routine can be mapped in the handler table in **usrHm.c** to a function pointer. There can be one callback routine per partition and one for the VxWorks 653 module. For details, see the *VxWorks 653 Configuration and Build Reference*.

If the function pointer is available, the callback routine is called whenever a partition event (or module event) arrives for the partition health monitor task (or module health monitor task). The callback routine is called before the handler.

8.3.5 Detecting and Reporting Application Errors

Applications are responsible for detecting their own errors and reporting (injecting) associated alarms.

An application can fail in such a way that it cannot correctly report the failure or cannot report a failure at all. System integrators may need to account for this possibility when they design the overall system.

Reporting for ARINC 653 Applications

For an application to conform to the ARINC 653 specification, it must use the APEX API. That is, to inject an alarm, it must issue the **RAISE_APPLICATION_ERROR** service and must report only the **APPLICATION_ERROR** ARINC 653-defined error. Other errors must be identified by potentially non-portable use of the service's message parameter.

The message that the **RAISE_APPLICATION_ERROR** service passes is read with the **GET_ERROR_STATUS** service. If the partition's error handler is created, it is then started to take the recovery action for the process that raises the error code. If the error handler is not created, the error is considered a partition-level error.

[Table 8-2](#) shows all the ARINC 653-defined errors, their numeric values, and equivalent health monitor alarm codes. If the application uses this service to inject any of the health monitor alarm codes or health monitor extended codes, the events are lost. The system integrator is responsible for preventing this.

ARINC 653 applications that want to handle a health monitor event code can first determine its equivalent ARINC 653 error by issuing the **GET_ERROR_STATUS** service.

If an ARINC 653 application wants to inject and handle health monitor alarm codes within the partition OS context, it would need to call **HM_EVENT_INJECT()**, **hmEventInject()**, or **hmErrorHandlerEventGet()**. Since this does not comply

with the ARINC 653 specification, the partition or module health monitor must handle them.

ARINC 653 Errors and Health Monitor Equivalents

Table 8-2 **ARINC 653 Errors and Their Health Monitor Alarm Code Equivalents**

ARINC 653 Error	Value	Health Monitor Alarm Code	Examples
APPLICATION_ERROR	1	HME_APPLICATION_ERROR	Errors raised by application processes.
DEADLINE_MISSED	0	HME_DEADLINE_MISSED	Process deadline violations.
HARDWARE_FAULT	6	HME_HARDWARE_FAULT	Memory-parity errors, I/O-access errors.
ILLEGAL_REQUEST	3	HME_ILLEGAL_REQUEST	Illegal OS request by a process.
MEMORY_VIOLATION	5	HME_MEMORY_VIOLATION	Memory-protection errors, supervisor privilege violations.
NUMERIC_ERROR	2	HME_NUMERIC_ERROR	Overflow errors, divide by zero, floating-point errors.
POWER_FAIL	7	HME_POWER_FAIL	Notification of power interruption so that, for example, application-specific state data can be saved.
STACK_OVERFLOW	4	HME_STACK_OVERFLOW	Process stack overflow.

Reporting for Non-ARINC 653 Applications

Applications that do not need to conform to the ARINC 653 specification can inject alarms by calling `HM_EVENT_INJECT()` or `hmEventInject()`. For more information about these routines, see [8.2.4 Injecting Alarms](#), p.202.

A partition that uses the APEX layer and wants to inject alarms within the partition OS context can inject an alarm by issuing `RAISE_APPLICATION_ERROR` with `APPLICATION_ERROR` or any of the other ARINC 653-defined codes in [Table 8-2](#).

8.4 Initializing the Health Monitor

The health monitor facility is initialized in the following stages:

1. The core OS initializes the module health monitor after it enables support for protection domains and before it initializes the ARINC 653 schedule.
2. As the core OS creates each partition, it initializes the associated partition health monitor after it assigns the partition to the proper window, but before it activates the partition.
3. If the application requested it, the core OS creates the error handler for the application.

8.5 Getting Health Monitor Information at Run-time

Configuration information is specified for the partition and module health monitors in the XML configuration file. For details, see the *VxWorks 653 Configuration and Build Guide*. User-supplied code in the core OS can get this information by calling `configRecordFieldGet()` with the appropriate `HM_TABLE_CFG_RECORD` configuration record (partition or module) and the appropriate field selector (defined in `configRecordLib.h`). The selectors are as follows:

- `HM_ATTRIBUTE_MASK`
- `HM_CALLBACK`

- HM_ENTRY_COUNT
- HM_ERROR_HANDLER_QUEUE_THRESHOLD
- HM_EVENT_CODE
- HM_EVENT_FILTER_MASK
- HM_HANDLER
- HM_LOG_ENTRIES_THRESHOLD
- HM_MAX_ERROR_HANDLER_QUEUE_DEPTH
- HM_MAX_LOG_ENTRIES
- HM_MAX_QUEUE_DEPTH
- HM_NOTIF_MAX_QUEUE_DEPTH
- HM_NOTIFICATION_HANDLER
- HM_QUEUE_THRESHOLD
- HM_STACK_SIZE
- HM_TRUSTED_PARTITION_MASK

8.6 Defining the Health Monitor Handler Table

The health monitor handler table, which is defined in the **usrHm.c** configlet, must define all handlers that are specified in the health monitor configuration. At initialization time, handler names in the table are resolved to function pointers.

8.6.1 Guidelines for Writing Handlers

Handlers should not make blocking calls without first making sure the alarm's injector cannot run until the health concern is fully handled. The health monitor facility assumes the handler is called synchronously within the context of the task or interrupt that injects the alarm. The facility dispatches the alarm to the appropriate level. In addition, it assumes the health monitor task at that level preempts the current context (the one that injected the alarm) or, if the alarm is injected from interrupt context, the handler is intentionally deferred. Thus, if the handler is pended due to a blocking call, the injecting context (if the injector is a task and not an interrupt handler) might be scheduled to run without having fully

handled the health concern that occurred in that task. In such a situation, it might be desirable to suspend the offending task before issuing a blocking call in the context of the handler.

Handlers must be located in the proper location for their level, as follows:

Type of Health Monitor Handler	Required Location
Process	Same partition OS that created the handler's context to run.
Partition	Kernel domain
Module	Kernel domain

8.7 Health Monitoring for COIL Partitions

(For information about COIL partitions, see [3. Developing COIL Applications](#).)

COIL provides an event API similar to what vThreads provides. Events injected by COIL-based applications are dispatched at the module level, partition level, or process level, in the same manner as for vThreads.

However, COIL-based applications get the configured dispatch level for an event by calling **coilHmEventInject()**. If the level is partition level (**HM_PARTITION_LVL**) or module level (**HM_MODULE_LVL**), the application need do nothing further. If the level is **HM_PROCESS_LVL**, the application must handle the event.

Example 8-1 Injecting a Health Monitor Event from a COIL-based Application

The following code fragment shows how to inject a health monitor event from a COIL-based application and detect its level. Constants are defined in the following file:

installDir/target/vThreads/h/hmTypes.h

```
int i;
int level = 0;
COIL_HM_EVENT event;
event.level = -1;
event.sysStatus = HM_PROCESS_EXEC_STATUS;
event.historicalCode = 0;
```

```
event.code = HM_MSG;
event.subCode = HM_SUB_CODE_STRING;
event.addInfo = 0;
event.addr = 0;
event.taskId = 1;
strcpy (event.taskName, "taskA");
strcpy (event.msg, "Message");
event.msgLen = strlen (event.msg);

coilHmEventInject(&event, 1, 1, &level);
if (level == HM_PROCESS_LVL )
{
    /* Handle event locally */
}
```

8.8 Other Facilities That Inject Alarms

The core OS injects alarms when conditions cause the system to restart.

The partition OS injects alarms under various conditions, such as when conditions cause the partition to restart.

The APEX layer injects alarms according to the ARINC 653 specification; for example, when a process misses its deadline.

Where possible and where appropriate, faults are mapped to a health monitor equivalent of an ARINC 653-defined code. This is especially true when handling the fault is possible or appropriate at the process level, such as in the case when applications generate exceptions.

8.9 Public Information

[Table 8-3](#) shows public health monitor information and where it is located. For information about a library and its routines, see their reference entries.

Table 8-3 Health Monitor Public Information

Type of Information	Location	Details
Header files (bring in the shared public header file <i>installDir/target/share/ h/hmTypes.h</i>	<i>installDir/target/h/hmLib.h</i>	Header file for the core OS.
	<i>installDir/target/vThreads/ h/hmLib.h</i>	Header file for vThreads applications.
	<i>installDir/target/val/h/ coilLib.h</i>	Header file for COIL-based applications.
Constants and data structures	hmTypes.h	<ul style="list-style-type: none">▪ System status fields and modes▪ Dispatch levels▪ ARINC 653-defined event codes▪ Wind River-defined event codes▪ Dispatch levels▪ Event subcodes
Core OS API	hmLib hmNotificationLib hmShow	
vThreads API	hmErrorHandlerLib hmLib hmShow	
COIL API	coilLib	
Default handlers	hmDefaultHandlers	Example health monitor handlers, available to module and partition health monitors.

Table 8-3 **Health Monitor Public Information** (cont'd)

Type of Information	Location	Details
	hmDbgDefaultHandlers	Example debug health monitor handlers, available to module and partition health monitors. Handlers try to suspend a task to allow debugging. When the task cannot be identified, the handlers suspend all tasks except the shell task, effectively shutting down the system to allow debugging.

9

I/O Support

- 9.1 Introduction 221
- 9.2 I/O and vThreads 221
- 9.3 Application Multiplexed I/O 256
- 9.4 I/O and COIL 264

9.1 Introduction

This chapter describes I/O support for vThreads partition OSs and a partition OSs based on COIL.

9.2 I/O and vThreads

An application performs synchronous I/O operations in a vThreads partition using the standard interface of **open()**, **close()**, **read()**, **write()**, and **ioctl()**. The POSIX AIO interface is available for asynchronous I/O operations. The POSIX AIO driver (**aioSysDrv**) issues read and write operations in the context of vThreads high-priority threads.

Applications can access devices created and managed by the core OS and devices created and managed within the vThreads partition (intrapartition devices).

A pipe device created by a thread in a partition is an example of an intrapartition device. Other threads in the partition can access the pipe device also. Intrapartition devices use the standard driver in the vThreads I/O system. In addition, a vThreads-based device driver manages a device namespace inherited from the core OS device namespace. The device driver handles all I/O requests from application threads to devices outside the partition.

The device driver uses the system-call mechanism (see [2.9 vThreads System Calls](#), p.34) to call core OS services to perform the I/O request. The mechanism allocates to the thread one of the core OS tasks (vThreads worker tasks, see [9.2.1 vThreads I/O and Worker Tasks](#), p.222) that is assigned to the partition. The thread (and only the thread) pends until the worker task completes the system call. Other threads in the partition continue to run. They can also perform I/O operations on an interpartition device. If that occurs, each thread is allocated a worker task to perform the I/O request when the interpartition device driver performs a system call.



NOTE: Although other threads in the partition do not block, the worker task performing the I/O call usually takes a global mutex. If the partition time slice completes before the I/O operation does, other partitions attempting to take the same mutex block until the first task completes, which will not be until the next partition time slice. An example of an I/O operation that may block is **printf()**.

The ANSI **stdio** facility (**fopen()**, **fclose()**, **fread()**, and **fwrite()**) uses the standard read and write routines to perform I/O. Therefore, no additional considerations are required to ensure that a blocking I/O operation does not stall the entire partition.

9.2.1 vThreads I/O and Worker Tasks

Each vThreads partition has a configurable number of worker tasks that are used to perform blocking operations. Worker tasks are core OS task that are associated with a given partition. They perform work on behalf of only their partition, in the time slot assigned to their partition. If worker tasks are not configured for a partition, all system calls run in the context of the partition, causing the partition to block until the call completes. Worker tasks are configurable when the **INCLUDE_DEBUG_UTIL** component is added to the core OS.

I/O operations might block even when all of the following are true:

- The `INCLUDE_DEBUG_UTIL` component is added to the core OS.
- The number of worker tasks is configured to non-zero.
- The `O_NONBLOCK` I/O flag is set when core OS-managed devices are opened. (For information on `O_NONBLOCK`, see [5.8.2 POSIX Message Queue Attributes](#), p.110.)

If the number of worker tasks in the partition is not enough to allocate a worker task to the I/O operation in every case, blocking might occur.

When the last worker task is used, a health monitor alarm is injected. When a system call cannot be dispatched because no worker tasks are available, a second alarm is injected.

Where I/O operations might block (as described above), do the following to ensure *no* I/O operations block:

- Add `INCLUDE_DEBUG_UTIL` to the core OS and configure a large enough number of worker tasks so that blocking of the I/O operation by a shortage of worker tasks does not occur.

Where I/O operations might block (as described above), do either of the following to ensure *all* I/O operations block:

- Exclude the `INCLUDE_DEBUG_UTIL` component from the core OS and do not create worker tasks for blocking system services.

or

- Configure the number of worker tasks to zero and do not create worker tasks for blocking system services.

9.2.2 Device Driver Models

Device drivers for vThreads partitions can follow one of the following models:

- located entirely in a vThreads partition (user mode)
- located entirely in the core OS (supervisor mode)
- split between the core OS and a vThreads partition (user mode and supervisor mode)

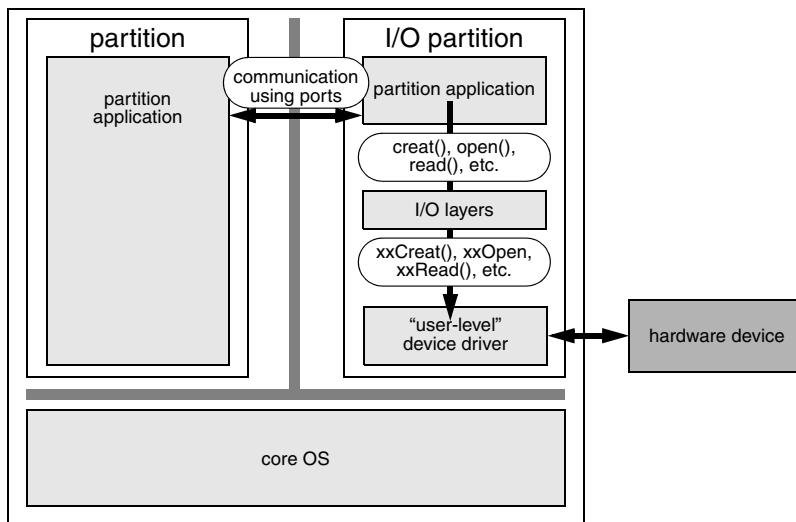
vThreads Model of Device Drivers

The vThreads model of device driver accesses the memory-mapped I/O registers directly from the I/O partition. The driver does not generate system calls to the core OS. In order for the I/O partition to read from and write to the memory-mapped I/O device, the I/O pool region must allow user-level access. This is best done by specifying the I/O space in a shared region that is accessible from the I/O partition only. The region map is specified at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*. Only the I/O partition can access the device directly. Other partitions communicate with the device through the I/O partition using an interpartition communication method.

Usually, non-I/O partitions rely on a local I/O device driver that uses ARINC 653 ports to communicate with the I/O partition. When the I/O partition is restarted, this device and its I/O layers are automatically reinitialized. For more information, see [7.9 Restart Functionality](#), p.156.

Because the vThreads model allows reading and writing of the I/O pool region from a partition, it is important that the region not cause bus errors or other critical errors that generate exceptions, which the core OS must handle.

Figure 9-1 vThreads Model of Device Driver



In the vThreads model, the device is controlled entirely by the partition, right down to accessing the physical hardware. As a result, the device driver, its data

structures, and the I/O system reside in the partition. The device hardware must be mapped and accessible from user mode for the model to work. No system calls are used to access the device.

Because partitions do not receive hardware interrupts, vThreads-based device drivers operate in polled mode only. They operate only during the partition schedule window.

Due to the intrinsic limitations of polling, it is possible to lose data input to the device. This is an issue only when data is received. Whether data is lost depends on the following:

- size of the read buffer of the device's FIFO buffer
- frequency of the I/O partition window
- configuration in the I/O partition
- speed of the device connection

While either the I/O partition or the task that is waiting for the data-receive in the I/O partition is blocked, device access is pended. During this time, the data might be over-written when the device's FIFO buffer becomes full. To improve the stability of polling drivers, do any of the following:

- Use a larger FIFO buffer.
- Schedule the partition to run more often.
- Use a slower connection.

The platform provider must configure the VxWorks 653 module and schedule the I/O partition properly to reduce the risk of data loss.

Communication between the device-owning partition and other partitions is done through ARINC 653 ports. For information on ports, see [4.7.4 Ports](#), p.70.

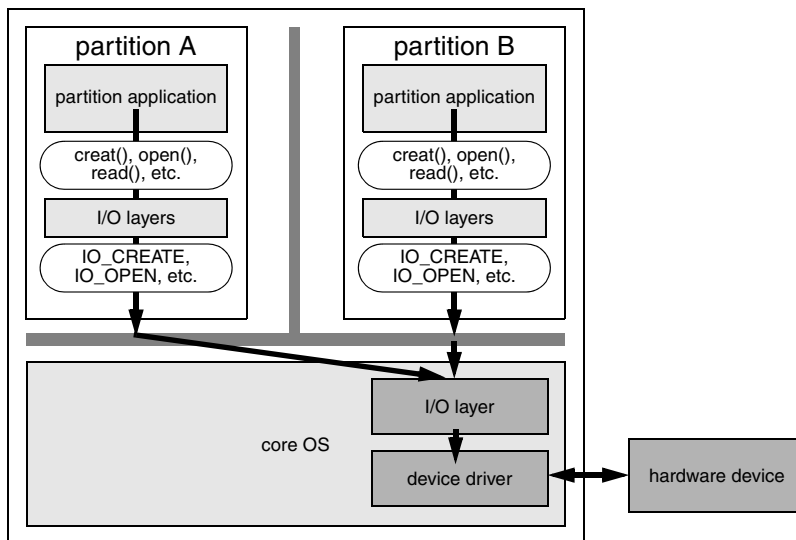
Core OS Model of Device Drivers

For the core OS model of device driver, the device driver is entirely in the core OS. The I/O pool region attribute must be set to allow supervisor-level access only. All partitions can access the device using the global **open()**, **close()**, **read()**, **write()**, and **ioctl()** routines. Interrupt and polling modes are supported. However, to respect ARINC scheduling, polling access to the device is strongly recommended.

When a partition restarts, the core OS issues the `ioctl() FIORESET` code to the file descriptor being accessed in order to force pending I/O system calls for the partition to complete. All the core OS file descriptors owned by the partition are closed. The `HME_POWER_FAIL` handler must initialize the core OS device drivers (except the system clock). The system integrator must ensure that the state of the core OS I/O layers is resynchronized with the new state of these device drivers.

The core OS model supports system warm restart, but not system cold restart.

Figure 9-2 Core OS Model of Device Driver



Split Model of Device Drivers

In the split model of device drivers, part of the driver is in the core OS and part is in a vThreads partition (usually an I/O partition). Only the I/O partition can access the device. Other partitions access the device by sending messages to the I/O partition using an ARINC 653 port.

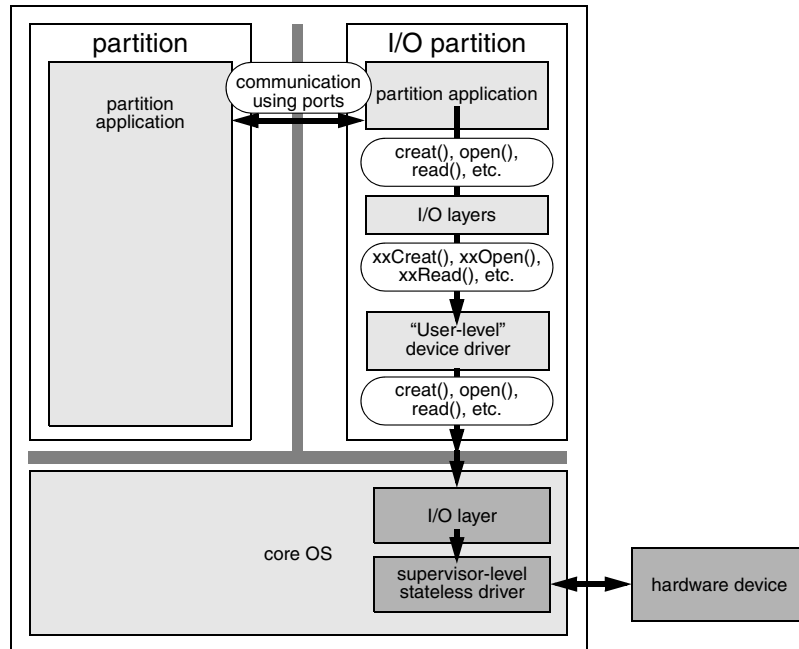
The core OS part of the driver accesses the device by generating a system call. Although interrupt-driven operation is possible in the core OS, to respect the time partitioning between the I/O partition and other partitions, polling access to the device is strongly recommended.

For a split-model driver, the I/O system (**ioLib**) and driver state information are in the partition. As a result, **open()**, **close()**, **read()**, **write()**, and so on, run in the partition. Since the device hardware is not mapped into the partition space, a system call is necessary to access the physical device. The core OS part of the driver need only implement rudimentary routines to access the device itself, along with routines to validate parameters. Because issues can arise when an I/O partition is restarted, the core OS part must be a stateless entity that does little more than access the physical device.

The split model accesses the I/O registers from an I/O partition through system calls to the core OS. Only the I/O partition can control the driver directly. Access to the device generates system calls that access routines in the core OS part. The I/O pool region attribute must be set to supervisor level. Other partitions access the device through the I/O partition using an interpartition communication method. Typically, non-I/O partitions rely on a local I/O device driver that uses ARINC 653 ports to communicate with the I/O partition.

When the I/O partition is restarted, the device and its I/O layers are also restarted. For more information, see [7.9 Restart Functionality](#), p.156.

Figure 9-3 Split Model of Device Driver



Validating the Read/Write Address Space

It is strongly recommended that the core OS portion of the driver validate the following for the read/write address space:

- page attributes
- strings
- partition buffers
- kernel buffer

For more information, see the reference entry for **valValidateLib**.

ioctl() code FIORESET Support

The split-model driver must support the **ioctl() FIORESET** code, which is called during partition restart if the **read()**, **write()**, or **ioctl()** operations on the device are not finished. (For more information, see [7.9.5 Restart Implications for Drivers](#), p.162.) Because the driver assumes that no events cause a blocking I/O operation, the only requirement is that **ioctl()** return OK.

9.2.3 Select Capability

A task can perform a select operation on file descriptors opened on vThreads (local) devices or core OS (global) devices.

Local select operations are those where a **select()** operation is done on a local file descriptor. Global select operations are performed on global file descriptors. The **fd_set** passed to **select()** can have a mixture of local and global file descriptors.

As part of the **selectLib** initialization, a select-server vThreads task (**tSelGblFdTask**) is spawned. The select server accepts global **select()** requests from vThreads tasks and performs global select operations serially on their behalf.

The select server uses a vThreads-wide global queue to serialize global **select()** operations done by multiple vThreads tasks. The vThreads **SELECT_SERVER_QSIZE** configuration parameter establishes the queue size and implies the concurrency level of global select operations in vThreads. Increasing the queue size correspondingly increases the number of tasks that can perform concurrent select operations on core OS file descriptors. The cost of increasing the parameter is 4 bytes per unit of increase. That is, each queue element is 4 bytes.

The select server uses a blocking system call to perform the global select operation. A single worker task for that partition is used to do the actual **select()** in the core OS.



NOTE: The concurrency level for the global select operation in vThreads space is equal to **SELECT_SERVER_QSIZE**. In the core OS space, it is always 1 because there is a single **select()** server task that accomplishes the select operation in the core OS.

Sample Drivers for Communicating Using ARINC 653 Ports

These code examples demonstrate how to communicate between the I/O partition and other partitions. The code is not included with the VxWorks 653 installation.

The communication mechanism requires the sets of ports listed in the following tables. You must add the ports to the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Guide*.

Table 9-1 Sampling Port Created in the I/O Partition

Name	Type	Source/Destination	Role
sMSync	Sampling, source	To all partitions	The I/O partition uses the port to send its status to other partitions.

One of the above is required for each VxWorks 653 module.

Table 9-2 Queuing Ports Created in the I/O Partition

Name	Type	Source/Destination	Role
qPDrvInPx	Queuing, source	To partition x	The I/O partition uses the port to send data read from the device that the I/O partition manages.
qPDrvOutPx	Queuing, destination	From partition x	The I/O partition uses the port to receive the type of request and output data.

One set of the above is required for each partition that communicates with the I/O partition.

Table 9-3 Sampling and Queuing Ports Created in Other Partitions

Name	Type	Source/Destination	Role
qPDrvInPx	Queuing, destination	From I/O partition	The data read from the device managed by the I/O partition is sent through the port.
qPDrvOutPx	Queuing, source	To I/O partition	The type of the request and the output data are sent through the port.
sMSyncPx	Sampling, destination	From I/O partition	I/O partition status is sent through the port.

One set of the above is required for each partition that communicates with the I/O partition.

If the I/O partition is Partition 1 and the partition communicating with it is Partition 2, only Partition 2 sends read and write requests to the I/O partition through the ports.

Example 9-1 **portRecords Structure**

```
LOCAL PORT_CFG_RECORD portRecords[] =
{
    /* The S port P1 to all partition SRC */
    { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "sMSync", 1, 1, 1,
      SOURCE, SAMPLING, NOT_APPLICABLE, 1, 0, 100000000, 0,0},

    /* The S port P1 to P2 DST */
    { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "sMSyncP2", 2, 1,
      1, DESTINATION, SAMPLING, NOT_APPLICABLE, 1, 0,
      100000000, 0,0},

    /* The Q port P2 to P1 SRC */
    { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvOutP2", 2, 1,
      2, SOURCE, QUEUING, SENDER_BLOCK, 256, 10, ZERO_TIME_VALUE ,0,0 },

    /* The Q port P2 to P1 DST */
    { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvOutP2", 1, 1,
      2, DESTINATION, QUEUING, NOT_APPLICABLE, 256, 10,
      ZERO_TIME_VALUE ,0,0 },

    /* The Q port P1 to P2 SRC */
    { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvInP2", 1, 1, 3,
      SOURCE, QUEUING, SENDER_BLOCK, 256, 10, ZERO_TIME_VALUE ,0,0 },
```



```

/* The Q port P1 to P2 DST */
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvInP2", 2, 1,
3, DESTINATION, QUEUING, NOT_APPLICABLE, 256, 10,
ZERO_TIME_VALUE ,0,0 },
};

```

Example 9-2 Partition I/O Handler Driver

This code example shows how to do the following:

- Establish communications between partitions and an I/O partition.
- Handle the read and write requests sent from non-I/O partitions through ARINC 653 ports.
- Call the TTY driver's **read()** and **write()** routines.
- Return the results using ARINC 653 ports.

Place the code in the following location:

installDir/target/vThreads/config/comps/src/usrPartHandleIO.c

```

/* usrPartHandleIO.c - stub partition I/O handler routine */

/* Copyright 2005      Wind River Systems, Inc. */

/*
DESCRIPTION
This is the source configlet for the INCLUDE_PART_IO_HANDLER component. It
creates two tasks that receive/send messages through queuing ports and
write/read consoleFd.

```

The partition I/O handler requires sMSync source sampling port, and qPDrvOutPx destination and qPDrvInPx source queuing ports for each partition domain that partition I/O device is installed.

for the I/O handler partition:

```

/* The S port Py to all partition SRC */
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "sMSync", y, 1, 1, SOURCE,
SAMPLING, NOT_APPLICABLE, 1, 0, 100000000, 0,0},

```

for each partition domain:

```

/* The Q port Px to Py DST */
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvOutPx", y, 1, 2,
DESTINATION, QUEUING, NOT_APPLICABLE, PART_DRV_QUEUE_MAX_MSG_SIZE,
PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0},

```

```

/* The Q port Py to Px SRC */
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvInPx", y, 1, 3,
SOURCE, QUEUING, SENDER_BLOCK, PART_DRV_QUEUE_MAX_MSG_SIZE,
PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0},

```

```

x = partition number I/O device is installed
y = partition number I/O handler is installed

```

the channel numbers (1, 2, and 3) may need to be adapted to your specific configuration

CONFIG PARAMETERS

The following parameters can be set and adjusted to alter the behaviour of this component.

PART_IO_DEV_FIRST_PARTITION

The first partition domain number that requires the I/O handler.

PART_IO_DEV_LAST_PARTITION

The last partition domain number that requires the I/O handler.

PART_DRV_TASK_PRI

The I/O handler task priority. The range is between 100 and 254.

PART_DRV_TASK_OPT

The I/O handler task option.

PART_DRV_TASK_STACK

The I/O handler task stack size.

PART_DRV_QUEUE_MAX_MSG_SIZE

The queuing port max message size. This must match the XML configuration file.

PART_DRV_QUEUE_MAX_NB_MSG

The queuing port max message number. This must match the XML configuration file.

PART_DRV_RCV_BUFF_SIZE

The receive buffer size.

*/

```
#include "taskLib.h"
#include "ioLib.h"
#include "string.h"
#include "stdio.h"
#include "apex/apexLib.h"
```

```
/* defines */
```

```
#define PART_SYNC_PORT_NAME          ("sMSync")
#define PART_OUT_PORT_BASE_NAME     ("qPDrvOutP")
#define PART_IN_PORT_BASE_NAME      ("qPDrvInP")
#define DRV_OUT_TASK_BASE_NAME      ("drvOutTaskP")
#define DRV_IN_TASK_BASE_NAME       ("drvInTaskP")
#define PART_DRV_MAX_MSG_SIZE       (PART_DRV_QUEUE_MAX_MSG_SIZE - 1)
#define READ_REQ_BYTES               readReqBytes [partNum - 1]
```

```
typedef enum PART_IO_REQUEST_CODE_TYPE /* request code type */
{
    PART_IO_READ,
    PART_IO_WRITE,
    PART_IO_SYNC,
    PART_IO_CANCEL
}
```

```

    } PART_IO_REQUEST_TYPE;

typedef enum PART_IO_STATUS_CODE_TYPE          /* status code type */
{
    PART_IO_SETUP,
    PART_IO_READY
} PART_IO_STATUS_TYPE;

/* externs */

IMPORT int consoleFd;

/* local */

LOCAL STATUS partIOHandlerInit (int);
LOCAL void partDrvOutputRtn (QUEUEING_PORT_ID_TYPE, QUEUEING_PORT_ID_TYPE, \
                             SEM_ID, int, char *);
LOCAL void partDrvInputRtn (QUEUEING_PORT_ID_TYPE, SEM_ID, int, char *);

/* global */

/* partition create status. The value remains after partition warm restart and
 * is re-initialized to FALSE at partition cold restart.
 */

LOCAL BOOL partCreate __attribute__((__section__(".persistent.data"))) = FALSE;

/* read request max size from PDx. The value remains after partition warm
 * restart and is cleared at partition cold restart.
 */

LOCAL int readReqBytes [PART_IO_DEV_LAST_PARTITION] \
    __attribute__((__section__(".persistent.bss")));

/*****
 *
 * usrPartIOHandlerInit--call the partition I/O handler initialization routine
 *
 * Calls the I/O handler initialization routine for each partition
 * domain that partition I/O device is installed. It is called automatically
 * by the root task, vThreadsCompInit(), in
 * prjConfig.c when the configuration macro INCLUDE_PART_IO_HANDLER is
 * defined.
 *
 * This routine requires SMSync source sampling port.
 *
 * /* The S port Py to all partition SRC */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "SMSync", y, 1, "", 0, 1, SOURCE,
 *   SAMPLING, NOT_APPLICABLE, 1, 0, 100000000, 0,0},
 *
 *   y = partition number I/O handler is installed
 *   the channel number (1) may need to be adapted to your configuration
 *
 * RETURNS: N/A.
 *****/

```

```
*/

void usrPartIOHandlerInit (void)
{
    SAMPLING_PORT_ID_TYPE    sMSyncId;        /* sMSync sampling port ID */
    RETURN_CODE_TYPE         retCode;
    int                      partNum;         /* partition number */
    char                     status;

    /* create a sampling port */

    CREATE_SAMPLING_PORT (PART_SYNC_PORT_NAME,
                          1,
                          SOURCE,
                          (SYSTEM_TIME_TYPE) 100000000,
                          &sMSyncId,
                          &retCode);

    if (retCode == NO_ACTION)
    {
        /* get sampling port ID if already attached */

        GET_SAMPLING_PORT_ID (PART_SYNC_PORT_NAME, &sMSyncId, &retCode);
    }

    if (retCode != NO_ERROR)
        return;

    /* write the status to sMSync and notice the partition I/O device
     * the handler is in setup
     */

    status = PART_IO_SETUP;

    WRITE_SAMPLING_MESSAGE (sMSyncId, &status, 1, &retCode);

    if (retCode != NO_ERROR)
        return;

    if (partCreate != TRUE)
    {
        /* if cold start, clean the read request */

        for (partNum = PART_IO_DEV_FIRST_PARTITION; \
             partNum <= PART_IO_DEV_LAST_PARTITION; partNum++)
            READ_REQ_BYTES = 0;

        partCreate = TRUE;
    }

    for (partNum = PART_IO_DEV_FIRST_PARTITION; \
         partNum <= PART_IO_DEV_LAST_PARTITION; partNum++)
    {
        /* Initialize I/O handler for each partition I/O device */

        partIOHandlerInit (partNum);
    }
}
```

```

    }

    /* write the status to SMSync and notice the partition I/O device
     * the handler is ready
     */

    status = PART_IO_READY;

    WRITE_SAMPLING_MESSAGE (SMSyncId, &status, 1, &retCode);

    if (retCode != NO_ERROR)
        return;
    }

/*****
 *
 * partIOHandlerInit - initialize the partition I/O handler
 *
 * This routine initializes the handler.
 *
 * This routine requires qPDrvOutPx and qPDrvInPx queuing ports.
 *
 * /* The Q port Px to Py DST */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvOutPx", y, 1,
 * 2, DESTINATION, QUEUING, NOT_APPLICABLE, PART_DRV_QUEUE_MAX_MSG_SIZE,
 * PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0},
 *
 * /* The Q port Py to Px SRC */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvInPx", y, 1, "", 0, 1,
 * SOURCE, QUEUING, SENDER_BLOCK, PART_DRV_QUEUE_MAX_MSG_SIZE,
 * PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE, 0,0},
 *
 * x = partition number I/O device is installed
 * y = partition number I/O handler is installed
 * the channel number (2) may need to be adapted to your configuration
 *
 * RETURNS: OK, or ERROR if fails.
 */

LOCAL STATUS partIOHandlerInit
(
    int                partNum        /* partition number */
)
{
    QUEUING_PORT_ID_TYPE    qPSendId;    /* qPDrvOutPx queuing port ID */
    QUEUING_PORT_ID_TYPE    qPRecvId;    /* qPDrvInPx queuing port ID */
    RETURN_CODE_TYPE        retCode;
    SEM_ID              semId;          /* binary semaphore ID */
    int                tid;
    char *              sendBuff;      /* transmit buffer pointer */
    char *              rcvBuff;      /* receive buffer pointer */
    char                objName [20];

    /* set name of qPDrvOutPx */

    sprintf (objName, "%s%d", PART_OUT_PORT_BASE_NAME, partNum);

```

```
/* create a queuing port */

CREATE_QUEUEING_PORT (objName,
                      PART_DRV_QUEUE_MAX_MSG_SIZE,
                      PART_DRV_QUEUE_MAX_NB_MSG,
                      DESTINATION,
                      FIFO,
                      &qPSendId,
                      &retCode);

if (retCode == NO_ACTION)
{
    /* get queuing port ID */

    GET_QUEUEING_PORT_ID (objName, &qPSendId, &retCode);
}

if (retCode != NO_ERROR)
    return (ERROR);

/* set name of qPDrvInPx */

sprintf (objName, "%s%d", PART_IN_PORT_BASE_NAME, partNum);

/* create a queuing port */

CREATE_QUEUEING_PORT (objName,
                      PART_DRV_QUEUE_MAX_MSG_SIZE,
                      PART_DRV_QUEUE_MAX_NB_MSG,
                      SOURCE,
                      FIFO,
                      &qPRecvId,
                      &retCode);

if (retCode == NO_ACTION)
{
    /* get queuing port ID */

    GET_QUEUEING_PORT_ID (objName, &qPRecvId, &retCode);
}

if (retCode != NO_ERROR)
    return (ERROR);

/* create counting semaphore */

semId = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

if ((sendBuff = malloc (PART_DRV_QUEUE_MAX_MSG_SIZE)) == NULL)
    return (ERROR);

if ((rcvBuff = malloc (PART_DRV_RCV_BUFF_SIZE + 1)) == NULL)
{
    free (rcvBuff);
    return (ERROR);
}
```

```

    }

    /* set name of drvOutTaskPx */

    sprintf (objName, "%s%d", DRV_OUT_TASK_BASE_NAME, partNum);

    /* cerate output task drvOutTaskPx */

    tid = taskSpawn (objName,
                     PART_DRV_TASK_PRI,
                     PART_DRV_TASK_OPT,
                     PART_DRV_TASK_STACK,
                     (FUNCPTR) partDrvOutputRtn,
                     qPSendId,
                     qPRecvId,
                     (int) semId,
                     partNum,
                     (int) sendBuff,
                     6, 7, 8, 9, 10);

    if (tid == ERROR)
    {
        free (sendBuff);
        free (rcvBuff);
        return (ERROR);
    }

    /* set name of drvInTaskPx */

    sprintf (objName, "%s%d", DRV_IN_TASK_BASE_NAME, partNum);

    /* cerate input task drvInTaskPx */

    tid = taskSpawn (objName,
                     PART_DRV_TASK_PRI + 1,
                     PART_DRV_TASK_OPT,
                     PART_DRV_TASK_STACK,
                     (FUNCPTR) partDrvInputRtn,
                     qPRecvId,
                     (int) semId,
                     partNum,
                     (int) rcvBuff,
                     5, 6, 7, 8, 9, 10);

    if (tid == ERROR)
    {
        free (rcvBuff);
        return (ERROR);
    }

    /* release semaphore if read request is in progress*/

    if (READ_REQ_BYTES != 0)

```

```

        semGive (semId);

    return (OK);
}

/*****
 *
 * partDrvOutputRtn - partition Output handler task for the x partition
 *
 * This task waits the I/O handling requests. If READ request is received,
 * release semaphore to unblock the partition Input handler task. If WRITE
 * request, write data to the I/O. If CANCEL request, call ioctl and cancel
 * the read request.
 *
 * RETURNS: N/A.
 */

LOCAL void partDrvOutputRtn
(
    QUEUING_PORT_ID_TYPE    qPSendId,    /* qPDrvOutPx queuing port ID */
    QUEUING_PORT_ID_TYPE    qPRecvId,    /* qPDrvInPx queuing port ID */
    SEM_ID                  semId,        /* binary semaphore ID */
    int                      partNum,      /* partition number */
    char *                   sendBuff      /* transmit buffer pointer */
)
{
    RETURN_CODE_TYPE         retCode;
    MESSAGE_SIZE_TYPE        msgLength;

    /* infinit loop */

    FOREVER
    {
        /* wait request messages from x partition */

        RECEIVE_QUEUING_MESSAGE (qPSendId,
                                INFINITE_TIME_VALUE,    /* wait forever */
                                sendBuff,
                                &msgLength,
                                &retCode);

        if (retCode == NO_ERROR)
        {
            /* check the request */

            switch (sendBuff [0])
            {
                case PART_IO_READ:                /* READ request */
                {
                    /* set read request bytes */
                    READ_REQ_BYTES = sendBuff [1] << 24 |
                                    sendBuff [2] << 16 |
                                    sendBuff [3] << 8 |
                                    sendBuff [4];

                    semGive (semId);                /* release semaphore */
                }
            }
        }
    }
}

```



```

        break;
    }

    case PART_IO_WRITE:                                /* WRITE request */
    {
        /* write the data to consoleFd */

        write (consoleFd, &sendBuff [1], msgLength - 1);

        break;
    }

    case PART_IO_SYNC:                                /* SYNC request */
    {
        /* send it back to x partition */

        SEND_QUEUEING_MESSAGE (qPRecvId,
                                sendBuff,
                                1,
                                INFINITE_TIME_VALUE,
                                &retCode);

        /* go through CANCEL request */
    }

    case PART_IO_CANCEL:                                /* CANCEL request */
    {
        if (READ_REQ_BYTES != 0)
        {
            /* if read is in progress, cancel it */

            READ_REQ_BYTES = 0;
            ioctl (consoleFd, FIORFLUSH, 0);
        }

        break;
    }

    default:                                            /* unexpected */
        break;
    }
}
}
}

/*****
 *
 * partDrvInputRtn - partition Input handler task for the x partition
 *
 * This task waits on the semaphore from Output handler task that is released
 * when READ request is received by the Output handler task and reads the I/O.
 *
 * RETURNS: N/A.
 */

```

```
LOCAL void partDrvInputRtn
(
    QUEUING_PORT_ID_TYPE    qPrcvId,        /* qPDrvInPx queuing port ID */
    SEM_ID                  semId,          /* binary semaphore ID */
    int                     partNum,        /* partition number */
    char *                  rcvBuff        /* message buffer */
)
{
    RETURN_CODE_TYPE        retCode;
    int                     maxbytes;      /* read max bytes */
    int                     msgLength;     /* message read bytes */
    int                     bytesPut;      /* message send bytes */
    char *                  buffer;

    /* infinite loop */

    FOREVER
    {
        /* wait for receive request from x partition */
        semTake (semId, WAIT_FOREVER);

        maxbytes = READ_REQ_BYTES;

        if (maxbytes > PART_DRV_RCV_BUFF_SIZE)
            maxbytes = PART_DRV_RCV_BUFF_SIZE;
            /* don not exceed PART_DRV_RCV_BUFF_SIZE */

        /* read consoleFd if any input arrives */
        msgLength = read (consoleFd, &rcvBuff [1], maxbytes);

        buffer = rcvBuff;

        if (msgLength == 0)                /* read canceled */
            retCode = NO_ERROR;

        while (msgLength > 0)
        {
            if (msgLength > PART_DRV_MAX_MSG_SIZE)
            {
                buffer [0] = TRUE;                /* continued */
                bytesPut = PART_DRV_MAX_MSG_SIZE;
            }
            else
            {
                buffer [0] = FALSE;                /* end */
                bytesPut = msgLength;
            }

            /* send it to x partition */
            SEND_QUEUEING_MESSAGE (qPrcvId,
                                   buffer,
                                   bytesPut + 1,
                                   INFINITE_TIME_VALUE,
                                   &retCode);

            if (retCode == NO_ERROR)
```

```

        {
            msgLength -= bytesPut;
            READ_REQ_BYTES = msgLength;

            buffer += bytesPut;
        }

/* unexpected */

if (retCode != NO_ERROR)
    semGive (semId);                /* release semaphore, read and send
                                    * the message again.
                                    */
    }
}

```

Example 9-3 Component Configuration File for the Partition I/O Handler

The following component configuration code specifies the configuration for the partition I/O handler driver in [Example 9-2](#). Place the code in the following location:

installDir/target/vThreads/config/comps/vxWorks/00comp_part_io_handler.cdf.

For information on how to install the partition I/O handler component, see the *VxWorks 653 Configuration and Build Guide*.

```

/* 00comp_part_io_handler.cdf - Component configuration file */

/* Copyright 2005      Wind River Systems, Inc. */

Component INCLUDE_PART_IO_HANDLER {
    NAME                Partition I/O Handler
    SYNOPSIS            Partition I/O Handler component
    REQUIRES            INCLUDE_APEX \
                        INCLUDE_SIO
    CONFIGLETTES        usrPartHandleIO.c
    _INIT_ORDER         vThreadsCompInit
    INIT_RTN            usrPartIOHandlerInit ();
    CFG_PARAMS          PART_IO_DEV_FIRST_PARTITION \
                        PART_IO_DEV_LAST_PARTITION \
                        PART_DRV_TASK_PRI \
                        PART_DRV_TASK_OPT \
                        PART_DRV_TASK_STACK \
                        PART_DRV_QUEUE_MAX_MSG_SIZE \
                        PART_DRV_QUEUE_MAX_NB_MSG \
                        PART_DRV_RCV_BUFF_SIZE
    PREF_DOMAIN         APPLICATION
}

Parameter PART_IO_DEV_FIRST_PARTITION {
    NAME                first partition number requires partition I/O
    DEFAULT             2
}

```

```
        TYPE          int
    }

Parameter PART_IO_DEV_LAST_PARTITION {
    NAME              last partition number requires part I/O
    DEFAULT           2
    TYPE              int
}

Parameter PART_DRV_TASK_PRI {
    NAME              IO handler task priority
    DEFAULT           100
    TYPE              int
}

Parameter PART_DRV_TASK_OPT {
    NAME              IO handler task option.
    DEFAULT           0
    TYPE              uint
}

Parameter PART_DRV_TASK_STACK {
    NAME              IO handler task stack size
    DEFAULT           0x1000
    TYPE              uint
}

Parameter PART_DRV_QUEUE_MAX_MSG_SIZE {
    NAME              partition queuing port max message size
    DEFAULT           256
    TYPE              int
}

Parameter PART_DRV_QUEUE_MAX_NB_MSG {
    NAME              partition queuing port max message number
    DEFAULT           10
    TYPE              int
}

Parameter PART_DRV_RCV_BUFF_SIZE {
    NAME              receive buffer size
    DEFAULT           255
    TYPE              int
}
```

Example 9-4 Driver to Communicate between I/O Partitions

The following code sample shows how to do the following:

- Establish communication between partitions and an I/O partition.
- Send the read and write requests to the I/O partition.
- Receive the results through ARINC 653 ports.

Place the code in the following location:

installDir/target/vThreads/config/comps/src/usrPartIODev.c

```
/* usrPartIODev.c - stub partition I/O driver file */
```

```
/* Copyright 2005      Wind River Systems, Inc. */
```

```
/*  
DESCRIPTION
```

This is the source configlet for the INCLUDE_PART_IO_DEV component.

The partition I/O device requires SMSyncPx sampling destination port and qPdrvOutPx source and qPdrvInPx destination queuing ports to synchronize and communicate to the partition I/O handler.

```
/* The S port P1 to Px DST */  
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "SMSyncPx", x, 1, 1,  
DESTINATION, SAMPLING, NOT_APPLICABLE, 1, 0, 100000000, 0,0},
```

```
/* The Q port Px to P1 SRC */  
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPdrvOutPx", x, 1, 2,  
SOURCE, QUEUING, SENDER_BLOCK, PART_DRV_QUEUE_MAX_MSG_SIZE,  
PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE, 0,0},
```

```
/* The Q port P1 to Px DST */  
{ CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPdrvInPx", x, 1, 3,  
DESTINATION, QUEUING, NOT_APPLICABLE, PART_DRV_QUEUE_MAX_MSG_SIZE,  
PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0},
```

x = PART_IO_PARTITION_NUMBER, y = partition number I/O handler is installed

CONFIG PARAMETERS

The following parameters can be set and adjusted to alter the behaviour of this component.

PART_IO_PARTITION_NUMBER
This partition's domain number.

PART_IO_SYNC_TIME_OUT_SEC
Synchronize time out period in seconds.

PART_DRV_QUEUE_MAX_MSG_SIZE
The queuing port max message size. This must match the XML configuration file.

PART_DRV_QUEUE_MAX_NB_MSG
The queuing port max message number. This must match the XML configuration file.
*/

```
#include "taskLib.h"  
#include "stdlib.h"  
#include "stdio.h"  
#include "string.h"
```

```
#include "errnoLib.h"
#include "iosLib.h"
#include "vThreads.h"
#include "apex/apexLib.h"

/* defines */

#define PART_IO_NAME ("/partIOP")
#define PART_SYNC_PORT_BASE_NAME ("sMSyncP")
#define PART_OUT_PORT_BASE_NAME ("qPDrvOutP")
#define PART_IN_PORT_BASE_NAME ("qPDrvInP")
#define PART_DRV_MAX_MSG_SIZE (PART_DRV_QUEUE_MAX_MSG_SIZE - 1)
#define REQUEST_BUFF_SIZE 5

typedef enum PART_IO_REQUEST_CODE_TYPE /* request code type */
{
    PART_IO_READ,
    PART_IO_WRITE,
    PART_IO_SYNC,
    PART_IO_CANCEL,
} PART_IO_REQUEST_TYPE;

typedef enum PART_IO_STATUS_CODE_TYPE /* status code type */
{
    PART_IO_SETUP,
    PART_IO_READY,
} PART_IO_STATUS_TYPE;

typedef struct /* PART_IO_DEV - partition I/O device descriptor */
{
    DEV_HDR devHdr; /* I/O device header */

    char rcvBuffer[PART_DRV_MAX_MSG_SIZE+1]; /* buffer for read */
    SEM_ID mutSemId; /* reader mutex semaphore */
    SAMPLING_PORT_ID_TYPE sMSyncId; /* synchronization port */
    QUEUING_PORT_ID_TYPE qPSendId; /* transmit port */
    QUEUING_PORT_ID_TYPE qPRecvId; /* reception port */
} PART_IO_DEV;

/* externs */

IMPORT int consoleFd;

/* forward declarations */

STATUS usrPartIODrv (void);
STATUS usrPartIODevCreate (char *);

/* locals */

LOCAL int usrPartIOOpen (PART_IO_DEV *, char *, int, int);
```

```

LOCAL int usrPartIOClose (PART_IO_DEV *);
LOCAL int usrPartIORead (PART_IO_DEV *, char *, int);
LOCAL int usrPartIOWrite (PART_IO_DEV *, char *, int);

LOCAL int partIODrvNum = ERROR;

/*****
 *
 * usrPartIOInit - initialize the partition I/O
 *
 * This routine calls the the driver install and initialization routines. It is
 * called automatically by the root task, vThreadsCompInit(), in prjConfig.c
 * when the configuration macro INCLUDE_PART_IO_DEV is defined.
 *
 * RETURNS: N/A.
 */

void usrPartIOInit (void)
{
    /* install the driver */

    if (usrPartIODrv () != OK)
        return;

    /* create the device */

    if (usrPartIODevCreate (PART_IO_NAME) != OK)
        return;

    /* open the device */

    consoleFd = open (PART_IO_NAME, O_RDWR, 0);

    ioGlobalStdSet (STD_IN, consoleFd);
    ioGlobalStdSet (STD_OUT, consoleFd);
    ioGlobalStdSet (STD_ERR, consoleFd);
}

/*****
 *
 * usrPartIODrv - initialize the part I/O driver
 *
 * This routine initializes and installs the driver.
 *
 * RETURNS: OK, or ERROR if the driver installation fails.
 */

STATUS usrPartIODrv (void)
{
    /* check if driver already installed */

    if (partIODrvNum != ERROR)
        return (OK);

```

```

    partIODrvNum = iosDrvInstall ((FUNCPTR) NULL, (FUNCPTR) NULL,
        usrPartIOOpen, usrPartIOClose, usrPartIORead,
        usrPartIOWrite, (FUNCPTR) NULL);

    return (partIODrvNum == ERROR ? ERROR : OK);
}

/*****
 *
 * usrPartIODevCreate - create a part I/O device
 *
 * This routine creates a part I/O device.
 *
 * This routine requires SMSyncPx sampling port, qPDrvOutPx and qPDrvInPx
 * queuing ports.
 *
 * /* The S port P1 to Px DST */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "SMSyncPx", x, 1, 1,
 * DESTINATION, SAMPLING, NOT_APPLICABLE, 1, 0, 100000000, 0,0},
 *
 * /* The Q port Px to P1 SRC */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvOutPx", x, 1, 2,
 * SOURCE, QUEUING, SENDER_BLOCK, PART_DRV_QUEUE_MAX_MSG_SIZE,
 * PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0 },
 *
 * /* The Q port P1 to Px DST */
 * { CFG_TYPE_PORT, sizeof (PORT_CFG_RECORD), "qPDrvInPx", x, 1, 3,
 * DESTINATION, QUEUING, NOT_APPLICABLE, PART_DRV_QUEUE_MAX_MSG_SIZE,
 * PART_DRV_QUEUE_MAX_NB_MSG, ZERO_TIME_VALUE ,0,0},
 *
 * x = PART_IO_PARTITION_NUMBER, y = partition number I/O handler is installed
 *
 * RETURNS: OK, or ERROR if the call fails.
 *
 * ERRNO
 * S_ioLib_NO_DRIVER - driver not initialized
 */

STATUS usrPartIODevCreate
(
    char *name          /* name of part I/O driver device to be created */
)
{
    PART_IO_DEV          *pPartIODrvDev;
    RETURN_CODE_TYPE     retCode;
    char                 objName [20];

    if (partIODrvNum == ERROR)
    {
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
    }

    pPartIODrvDev = (PART_IO_DEV *) malloc (sizeof (PART_IO_DEV));

```



```

if (pPartIODrvDev == NULL)
    return (ERROR);

/* I/O device to system */

if (iosDevAdd (&pPartIODrvDev->devHdr, name, partIODrvNum) != OK)
{
    free ((char *) pPartIODrvDev);
    return (ERROR);
}

/* set name of sMSyncPx */

sprintf (objName, "%s%d", PART_SYNC_PORT_BASE_NAME, \
        PART_IO_PARTITION_NUMBER);

/* create a sampling port */

CREATE_SAMPLING_PORT (objName,
                      1,
                      DESTINATION,
                      (SYSTEM_TIME_TYPE) 100000000,
                      &pPartIODrvDev->sMSyncId,
                      &retCode);

if (retCode == NO_ACTION)
{
    /* get sampling port ID */

    GET_SAMPLING_PORT_ID (objName,
                          &pPartIODrvDev->sMSyncId,
                          &retCode);
}

if (retCode != NO_ERROR)
{
    free ((char *) pPartIODrvDev);
    return (ERROR);
}

/* set name of qPDrvOutPx */

sprintf (objName, "%s%d", PART_OUT_PORT_BASE_NAME, \
        PART_IO_PARTITION_NUMBER);

/* create a queuing port */

CREATE_QUEUING_PORT (objName,
                     PART_DRV_QUEU_MAX_MSG_SIZE,
                     PART_DRV_QUEU_MAX_NB_MSG,
                     SOURCE,
                     FIFO,
                     &pPartIODrvDev->qPSendId,
                     &retCode);

if (retCode == NO_ACTION)

```

```
    {
        /* get queuing port ID */
        GET_QUEUING_PORT_ID (objName,
                             &pPartIODrvDev->qPSendId,
                             &retCode);
    }

    if (retCode != NO_ERROR)
    {
        free ((char *) pPartIODrvDev);
        return (ERROR);
    }

    /* set name of qPDrvInPx */

    sprintf (objName, "%s%d", PART_IN_PORT_BASE_NAME, PART_IO_PARTITION_NUMBER);

    /* create a queuing port */

    CREATE_QUEUING_PORT (objName,
                         PART_DRV_QUEUE_MAX_MSG_SIZE,
                         PART_DRV_QUEUE_MAX_NB_MSG,
                         DESTINATION,
                         FIFO,
                         &pPartIODrvDev->qPRecvId,
                         &retCode);

    if (retCode == NO_ACTION)
    {
        /* get queuing port ID */

        GET_QUEUING_PORT_ID (objName,
                             &pPartIODrvDev->qPRecvId,
                             &retCode);
    }

    if (retCode != NO_ERROR)
    {
        free ((char *) pPartIODrvDev);
        return (ERROR);
    }

    /* create a mutex semaphore */

    pPartIODrvDev->mutSemId = semMCreate (SEM_DELETE_SAFE);

    return (OK);
}

/*****
 *
 * usrPartIODevDelete - delete a part I/O device
 *
 * Deletes a part I/O device of a given name. The name must match
 * that passed to usrPartIODevCreate() else ERROR will be returned. This
 * routine frees memory for the necessary structures and deletes the device.
 *****/
```

```

*
* RETURNS: OK, or ERROR if the call fails.
*
* ERRNO
* S_ioLib_NO_DRIVER          - driver not initialized
*/

STATUS usrPartIODevDelete
(
    char *name                /* name of part I/O driver device to be deleted */
)
{
    PART_IO_DEV               *pPartIODrvDev;
    char *pTail = NULL;

    if (partIODrvNum == ERROR)
    {
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
    }

    if ((pPartIODrvDev = (PART_IO_DEV *) iosDevFind (name, &pTail))
        == NULL)
    {
        return (ERROR);
    }

    /* I/O device no longer in system */

    iosDevDelete (&pPartIODrvDev->devHdr);

    /* delete the semaphore */

    semDelete (pPartIODrvDev->mutSemId);

    /* free part I/O memory */

    free ((char *)pPartIODrvDev);

    return (OK);
}

/*****
*
* usrPartIOOpen - open a part I/O file
*
* This routine is called to open a part I/O file. It returns a pointer to the
* device.
*
* RETURNS: pPartIODrvDev or ERROR if time out synchronize the partition I/O
* handler.
*/

LOCAL int usrPartIOOpen
(
    PART_IO_DEV * pPartIODrvDev,                /* device to control */

```

```
char *      name,                /* device name */
int         flags,               /* flags */
int         mode                 /* mode selected */
)
{
MESSAGE_SIZE_TYPE msgLength;    /* length received */
VALIDITY_TYPE     validity;     /* validity for sync port */
RETURN_CODE_TYPE  retCode;
int              n100mSec;      /* x 100m sec */
char             message;       /* message */

n100mSec = 0;                   /* set time 0 */

FOREVER
{
    /* synchronize the partition I/O handler */

    READ_SAMPLING_MESSAGE (pPartIODrvDev->sMSyncId,
                           &message,
                           &msgLength,
                           &validity,
                           &retCode);

    if ((retCode == NO_ERROR) && (msgLength > 0) && \
        (message == PART_IO_READY))
        break;

    if (n100mSec < PART_IO_SYNC_TIME_OUT_SEC * 10) /* time out 5 sec */
        taskDelay (sysClkRateGet()/10);
    else
        return (ERROR);

    n100mSec++;
}

FOREVER
{
    message = PART_IO_SYNC;      /* SYNC request */

    /* send a SYNC request. It will cancel the previous READ request */

    SEND_QUEUEING_MESSAGE (pPartIODrvDev->qPSendId,
                           &message,
                           1,
                           INFINITE_TIME_VALUE,
                           &retCode);

    if (retCode != NO_ERROR)
        return (ERROR);

    /* receive SYNC message from partition I/O */

    RECEIVE_QUEUEING_MESSAGE (pPartIODrvDev->qPRecvId,
                              (SYSTEM_TIME_TYPE) 1000000000, /* 1 sec */
                              pPartIODrvDev->rcvBuffer,
                              &msgLength,
```

```

                                &retCode);

    if (retCode == NO_ERROR)
    {
        if ((msgLength == 1) && \
            (pPartIODrvDev->rcvBuffer [0] == PART_IO_SYNC))
            break;
        else
            return (ERROR);
    }
    else if (retCode == TIMED_OUT)
    {
        if (n100mSec < PART_IO_SYNC_TIME_OUT_SEC * 10) /* time out 5 sec */
            n100mSec += 10;
        else
            return (ERROR);
    }
    else
        return (ERROR);
}

return ((int) pPartIODrvDev);
}

/*****
 *
 * usrPartIOClose - close a part I/O file
 *
 * This routine is called to close a part I/O file.
 *
 * RETURNS: pPartIODrvDev or ERROR if NULL part I/O device pointer.
 */

LOCAL int usrPartIOClose
(
    PART_IO_DEV *pPartIODrvDev                /* device to control */
)
{
    RETURN_CODE_TYPE    retCode;
    char message;                                /* message for cancel request */

    if (pPartIODrvDev != NULL)
    {
        message = PART_IO_CANCEL;                /* CANCEL request */

        /* send a CANCEL request */

        SEND_QUEUEING_MESSAGE (pPartIODrvDev->qPSendId,
                                &message,
                                1,
                                INFINITE_TIME_VALUE,
                                &retCode);

        if (retCode != NO_ERROR)
            return (ERROR);
    }
}

```

```
        return ((int) pPartIODrvDev);
    }
    else
        return (ERROR);
    }

/*****
 *
 * usrPartIORead - read a partition I/O file
 *
 * This routine is called to read a part I/O file.
 * It reads into the buffer up to <maxbytes> available bytes.
 *
 * RETURNS: The number of bytes actually read into the buffer.
 */

LOCAL int usrPartIORead
(
    PART_IO_DEV *    pPartIODrvDev,          /* device to control */
    char *          buffer,                  /* buffer to read into */
    int             maxbytes                 /* maximum length of read */
)
{
    RETURN_CODE_TYPE  retCode;
    MESSAGE_SIZE_TYPE msgLength;
    int               bytesRcvd;             /* total received bytes */
    char              reqBuff [REQUEST_BUFF_SIZE];

    bytesRcvd = 0;                          /* clear the bytesRcvd */

    if (maxbytes > 0)
    {
        /* block receive process from other tasks */

        semTake (pPartIODrvDev->mutSemId, WAIT_FOREVER);

        reqBuff [0] = PART_IO_READ;          /* READ request */

        /* set max receive bytes */

        reqBuff [1] = (maxbytes & 0xff000000) >> 24;
        reqBuff [2] = (maxbytes & 0x00ff0000) >> 16;
        reqBuff [3] = (maxbytes & 0x0000ff00) >> 8;
        reqBuff [4] = (maxbytes & 0x000000ff);

        /* send a READ request */

        SEND_QUEUEING_MESSAGE (pPartIODrvDev->qPSendId,
                               reqBuff,
                               REQUEST_BUFF_SIZE,
                               INFINITE_TIME_VALUE,
                               &retCode);

        if (retCode == NO_ERROR)
        {
            pPartIODrvDev->rcvBuffer [0] = TRUE;
        }
    }
}
```

```

/* loop if continued status */
while (pPartIODrvDev->rcvBuffer [0] == TRUE)
{
    /* receive a message */

    RECEIVE_QUEUEING_MESSAGE (pPartIODrvDev->qPrcvId,
                              INFINITE_TIME_VALUE, /* wait forever */
                              pPartIODrvDev->rcvBuffer,
                              &msgLength,
                              &retCode);

    if ((retCode != NO_ERROR) || !(msgLength > 0))
    {
        /* release semaphore */

        semGive (pPartIODrvDev->mutSemId);
        return (bytesRcvd);
    }

    msgLength -= 1;                /* adjust received length */

    /* copy the received message to buffer */

    bcopy (&pPartIODrvDev->rcvBuffer [1], buffer, msgLength);

    maxbytes -= msgLength;
    bytesRcvd += msgLength;
    buffer += msgLength;
}

/* release semaphore */

semGive (pPartIODrvDev->mutSemId);
}

return (bytesRcvd);
}

/*****
 *
 * usrPartIOWrite - write a partition I/O file
 *
 * This routine is called to write a partition I/O file.
 *
 * RETURNS: The number of bytes actually written to the device.
 */

LOCAL int usrPartIOWrite
(
    PART_IO_DEV *    pPartIODrvDev,        /* device to control */
    char *           buffer,                /* buffer of data to write */
    int              nbytes                 /* number of bytes in buffer */
)

```

```
{
RETURN_CODE_TYPE  retCode;
int               bytesSend;      /* total bytes sent */
int               bytesPut;       /* bytes message sent through port */
char              sendBuffer [PART_DRV_MAX_MSG_SIZE + 1];

bytesSend = 0;                      /* clear the bytesSend */

/* loop till all messages sent through the port */

while (nbytes > 0)
{
    sendBuffer [0] = PART_IO_WRITE;      /* WRITE request */

    if (nbytes > PART_DRV_MAX_MSG_SIZE)
        bytesPut = PART_DRV_MAX_MSG_SIZE;
    else
        bytesPut = nbytes;

    /* copy the message to transmit buffer */

    bcopy (buffer, &sendBuffer [1], bytesPut);

    /* send a message with WRITE request */

    SEND_QUEUEING_MESSAGE (pPartIODrvDev->qPSendId,
                           sendBuffer,
                           bytesPut + 1,
                           INFINITE_TIME_VALUE,
                           &retCode);

    if (retCode == NO_ERROR)
    {
        nbytes -= bytesPut;
        bytesSend += bytesPut;
        buffer += bytesPut;
    }
    else
        break;
}

return (bytesSend);
}
```

Example 9-5 Component Configuration File for the Driver to Communicate between I/O Partition and Non-I/O Partitions

The following component configuration code explains how to configure the driver to communicate between I/O partition and non-I/O partitions in [Example 9-4](#).

Place the code in the following location:

installDir/target/vThreads/config/comps/vxWorks/00comp_part_io_dev.cdf

For information on how to include a partition I/O device component, see the *VxWorks 653 Configuration and Build Guide*.

```

/* 00comp_part_io_dev.cdf - Component configuration file */

/* Copyright 2005      Wind River Systems, Inc. */

Component INCLUDE_PART_IO_DEV {
    NAME                Partition I/O Device
    SYNOPSIS            Partition I/O Device component
    REQUIRES            INCLUDE_APEX
    CONFIGLETTES        usrPartIODev.c
    _INIT_ORDER         vThreadsCompInit
    INIT_RTN            usrPartIOInit ();
    CFG_PARAMS          PART_IO_PARTITION_NUMBER \
                        PART_IO_SYNC_TIME_OUT_SEC \
                        PART_DRV_QUEUE_MAX_MSG_SIZE \
                        PART_DRV_QUEUE_MAX_NB_MSG
    PREF_DOMAIN         APPLICATION
    ENTRY_POINTS        usrPartIODrv \
                        usrPartIODevCreate
}

EntryPoint usrPartIODrv {
    TYPE                TEXT
}

EntryPoint usrPartIODevCreate {
    TYPE                TEXT
}

Parameter PART_IO_PARTITION_NUMBER {
    NAME                partition number to be installed
    DEFAULT             2
    TYPE                int
}

Parameter PART_IO_SYNC_TIME_OUT_SEC {
    NAME                IO port synchronous time out sec
    DEFAULT             5
    TYPE                int
}

Parameter PART_DRV_QUEUE_MAX_MSG_SIZE {
    NAME                partition queuing port max message size
    DEFAULT             256
    TYPE                int
}

Parameter PART_DRV_QUEUE_MAX_NB_MSG {
    NAME                partition queuing port max message number
    DEFAULT             10
    TYPE                int
}

```

Table 9-4 shows where the drivers can be placed.

Table 9-4 **Components and Associated Domains**

Component	Where Can it Go?
INCLUDE_PART_IO_HANDLER	I/O partition
INCLUDE_PART_IO_DEV	Other partitions, but not the partition in which INCLUDE_PART_IO_HANDLER is installed

Supervisor-Level Device Driver Model

For information on available routines and ioctl commands, see the reference entry for the core OS **ioLib**.

9.3 Application Multiplexed I/O

Application multiplexed I/O provides application developers with a communications channel to output text to and receive responses from a dedicated maintenance or test terminal. The feature could be used for running automated test scripts or for debugging.

Application multiplexed I/O runs over a single, dedicated serial communications channel operating in polled mode. All partitions in the VxWorks 653 module share the same serial line for sending and receiving text. All data is sent in ASCII format with a header that indicates to which partition the data is assigned.

Application multiplexed I/O consists of two drivers: one in the partition and one in the core OS. The driver in the partition provides blocking read and write operations that do not block the entire partition. The driver in the core OS provides multiple partitions access to the shared serial I/O device.

Data received from or sent to the host is written into circular FIFO buffers. There is one input buffer and one output buffer per partition, plus a pair for the core OS. For read operations, data is overwritten when the input buffer is full. After data is overwritten, a read operation is still FIFO and reads the 'new' oldest data. That is, it reads the data available just after the overwritten data.

For application multiplexed I/O, the core OS allocates each partition a dynamic bandwidth that is based on the duration of the partition's time window and the bandwidth of the serialized I/O driver. Thus, for any time window, a partition that uses application multiplexed I/O can send and receive only a limited number of bytes. As a result, the partition's application multiplexed I/O does not affect other partitions. Throughput is proportional to the CPU time that the core OS allocates to the partition.

9.3.1 Serialized I/O Protocol

Application multiplexed I/O can use any serialized I/O driver, such as a terminal driver, pseudo-terminal, or pipe driver. The supported serialized I/O has the following protocol in both directions:

- The flow supports eight-bit data only.
- Each channel in the VxWorks 653 module has a unique channel ID.
- The channel ID is coded by two hexadecimal ASCII characters: 256 values from 0 to FF.
- Channel 0 identifies the core OS.
- Channels 1 through 255 identify partitions 1 through 255.
- Only the transition from one channel to another is identified, which is identified by this character sequence:

STX + channelID1 + channelID2 + data

- The link partner is made aware of system restart by this character sequence:
STX + NULL + NULL
- The ownership for each direction of the channel is independent. For example, the host can write to one partition while another partition is writing to the host.
- Data associated with an invalid channel ID is discarded.
- The target requests a new baud setting by sending:

STX + 0x01 + BRI

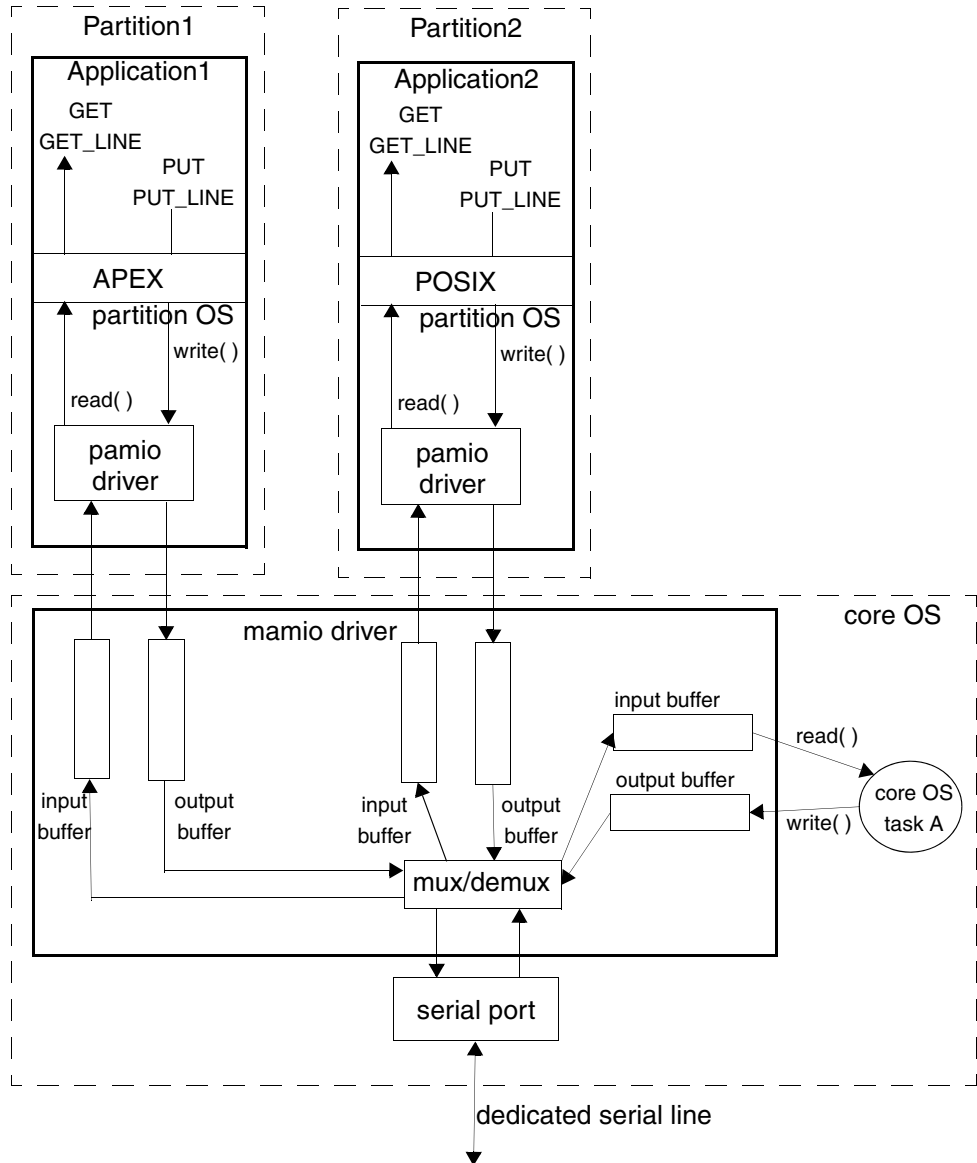
where *BRI* is encoded as follows:

BRI	Baud (in bps)
0x00	default
0x01	9600
0x02	9200
0x03	38400
0x04	57600
0x05	76800
0x06	115200

9.3.2 Architecture

[Figure 9-4](#) illustrates the general architecture of application multiplexed I/O.

Figure 9-4 Application Multiplex I/O Architecture



9.3.3 Setting up and Using Application Multiplexed I/O in Partitions

Making the Driver Available

If the optional **INCLUDE_AMIO** component is included in the application's partition, application multiplexed I/O is available to the application.

Redirecting Standard I/O to the **pamio** Driver

In order for standard **open()**, **read()**, **write()**, and **ioctl()** to use the **pamio** driver, standard I/O must be redirected to the driver.

If the **INCLUDE_AMIO_REDIRECT** component is included in the partition or one of its shared libraries, standard I/O is automatically redirected.

Alternatively, and for increased granularity, the application can redirect I/O by calling **ioGlobalStdSet()**, which overrides the standard file descriptor and causes the **pamio** file descriptor to be used instead. Example code follows:

```
int pamioFd = open ("/pamio", O_RDWR, 0);
ioGlobalStdSet (0, pamioFd); /* STD IN */
ioGlobalStdSet (1, pamioFd); /* STD OUT */
ioGlobalStdSet (2, pamioFd); /* STD ERROR */
```

Using Application Multiplexed I/O

The API for the **pamio** driver uses the standard VxWorks 653 system calls:

- **SYSCALL_IO_OPEN**
- **SYSCALL_IO_READ**
- **SYSCALL_IO_WRITE**
- **SYSCALL_IO_IOCTL**

The **read()** and **write()** routines are blocking. For example, if a channel's input buffer is empty, **read()** waits until an ASCII character is available. Each read or write operation is normally equivalent to a system call and subsequent read or write operation in the core OS. To avoid this potential inefficiency when there are no characters to read or no space to write to, the **pamio** driver blocks the requesting task. When characters are available for reading or space for writing, the **pamio** driver in the core OS sends a pseudo-interrupt to the partition, and the **pamio** driver unblocks the requesting task.

The **ioctl()** routine takes the following command codes:

FIOGETOPTIONS

Gets the current device-option word.

FIORELINQUISH

Frees all resources (the blocking semaphore) for the stopped task.

FIOSETOPTIONS

Sets the device-option word to the specified value.

9.3.4 Using Application Multiplexed I/O in the Core OS

The core OS driver that supports application multiplexed I/O in partitions (the **mamio** driver) is installed automatically when the kernel calls the **mamioLibInit()** routine. (The term **mamio** stands for module application multiplexed I/O.) The routine returns a **mamio** device file descriptor. The **mamio** driver follows the standard open, read, write, ioctl I/O model.

For information on how the core OS can use application multiplexed I/O for all I/O in the core OS, see [Using the mamio Driver for All I/O in the Core OS](#), p.263.

Setting the Mux/Demux Algorithm

When the VxWorks 653 module is cold started, **mamioLibInit()** calls **muxDemuxIOLibInit()**, which must be called after the **mamio** driver is initialized, but before any other **mamio** driver calls are made. The routine sets the polling period. In addition, it initializes the mux/demux algorithm according to these options:

MUX_BLOCKING_DEVICE

If this option is used, the mux write routine uses a **FIONWRITE** control operation on the serialized output device before writing a character. The mux read routine uses a **FIONREAD** control operation on the serialized output device before reading a character.

MUX_KERNEL_ALL_WINDOW

If this option is used, the core OS sends data during the time available from any partition, including ones that do not include the application multiplexed I/O component.

Since the core OS by default sends data only during spare partition windows, the platform provider must take care when configuring the system. That is, if the default mux/demux option is not changed, the platform provider must

include at least one spare time window in the schedule if the core OS uses the **mamio** driver.

MUX_KERNEL_SPARE_AND_AMIO_WINDOW

If this option is used, the core OS sends data during the time available from any partition that includes the application multiplexed I/O component. Core OS operations preempt the partition.

Using the ioctl() Routine

The **ioctl()** routine takes the following command codes:

FIOBAUDRATE

Sets the baud of the associated serialized device (same as **SIO_BAUD_SET**).

FIOGETOPTIONS

Gets the current device option word. For more information, see [Setting Line Modes](#), p.263.

FIONREAD

Returns the number of characters that are available to be read from the specified partition's input buffer. However, if the **MAMIO_OPTIONS_CRMOD** option is enabled, **ioctl()** returns the number of characters copied plus the number of lines in the buffer. For example, if five lines of **NEWLINES** are in the input buffer, the routine returns 10. For more information, see [Setting Line Modes](#), p.263.

FIONWRITE

Returns the number of characters queued to the specified partition's output buffer.

FIOSETOPTIONS

Sets the device option word to the specified value. For more information, see [Setting Line Modes](#), p.263.

MAMIO_IOCTL_BUFF_INPUT_PUT

Puts the specified number of characters from the specified buffer into the specified partition's input buffer. If the **MAMIO_OPTIONS_CRMOD** option is enabled, the number of characters that are put might be different from that specified. For more information, see [Setting Line Modes](#), p.263.

MAMIO_IOCTL_BUFF_OUTPUT_GET

Gets characters from the specified partition's output buffer and puts them in the specified buffer. The number of characters that are gotten is returned.

MAMIO_IOCTL_NOTICE_REGISTER

Registers the notification routine that is to be called when characters are available in the channel's output buffer or when the channel requests data to read.

SIO_BAUD_GET

Gets the baud of the associated serialized device.

SIO_BAUD_SET

Sets the baud of the associated serialized device (same as FIOBAUDRATE).

Setting Line Modes

If the **MAMIO_OPTIONS_CRMOD** option is enabled for the **mamio** driver's option word, the following occurs:

This character:	Is replaced with:
Received CR (<code>'\r'</code> or <code>0xD</code>)	NEWLINE (<code>'\n'</code> or <code>0xA</code>)
Sent NEWLINE	CR + NEWLINE

If the **MAMIO_OPTIONS_CRMOD** option is not enabled, no replacement is made.

If the **MAMIO_OPTIONS_LINE** option is enabled for the **mamio** driver's option word, the input character stream is not available for reading until a **NEWLINE** character or 255 characters are received. In addition, the input may be modified by the special characters of backspace (`0x8`), line-delete (`0x15`), and end-of-file (`0x4`). If the option is not enabled, these special characters are not removed.

Using the mamio Driver for All I/O in the Core OS

To have all I/O in the core OS be redirected to the **mamio** driver, calls must be made to **ioGlobalStdSet()**, which override the standard file descriptors and causes the **mamio** file descriptors to be used instead. Example code follows:

```
int mamioFd = open ("/mamio", O_RDWR, 0);
ioGlobalStdSet (0, mamioFd); /* STD IN */
ioGlobalStdSet (1, mamioFd); /* STD OUT */
ioGlobalStdSet (2, mamioFd); /* STD ERROR */
```

The above is not required for partitions to use application multiplexed I/O.

9.4 I/O and COIL

(For information about the core OS interface library, see [3. Developing COIL Applications](#).)

An application that runs in a partition with a partition OS based on COIL has available an API for device I/O system calls. The application has access to kernel device drivers, including user-supplied kernel device drivers.

COIL supplies a COIL API for I/O. These routines use the same device name strings as the vThreads device I/O routines and use them in the same manner.

Blocking Versus Non-blocking I/O (Compared to vThreads)

Both vThreads and COIL support blocking and non-blocking I/O system calls. However, vThreads abstracts the details so that the caller is not aware of whether the call blocks or whether the I/O work is handed off to a worker task. This non-blocking I/O infrastructure is not provided in COIL, and so the user partition OS must provide non-blocking I/O if it is needed.

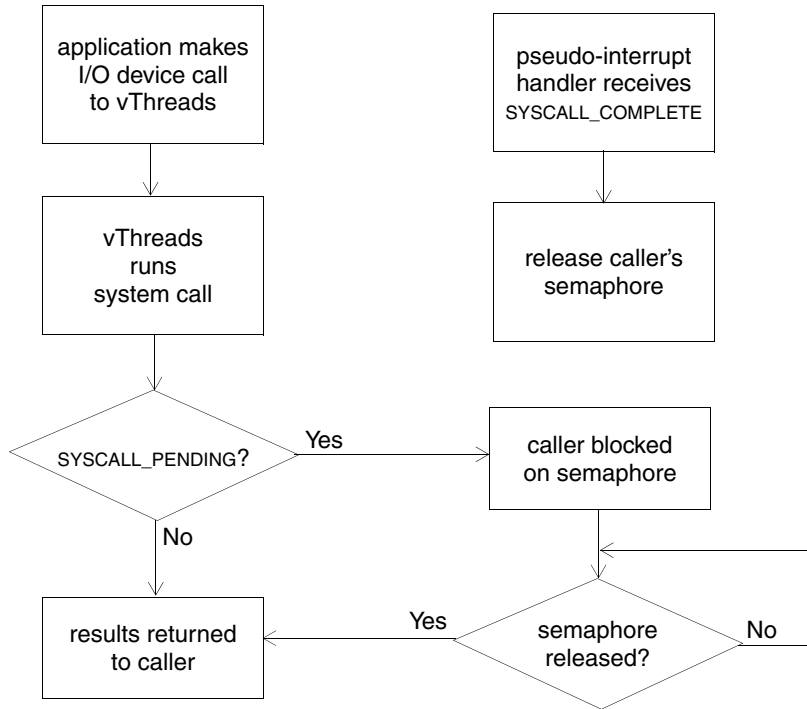
The blocking I/O case is simple, since the system call simply does not return until the device I/O activity has been completed and the results are returned. This occurs identically in both a vThreads partition and a partition based on COIL.

Consider the case where worker tasks are present and an application in a COIL-based partition makes a device I/O call. The call returns immediately to the caller with a return value of **COIL_SYSCALL_PENDING**. It is then up to the application to determine what to do until the device I/O work is complete. (If a vThreads partition makes the call, the calling thread is blocked on a semaphore and, therefore, does not return to the calling application.)

When the device I/O operation completes, a **COIL_EVENT_SYSCALL_COMPLETE** pseudo-interrupt is delivered to the partition. (In a vThreads partition, this pseudo-interrupt releases the calling thread from its semaphore and lets it return to the caller.) In the case of a partition OS based on COIL, it is up to the user-provided pseudo-interrupt handler routine to determine how to proceed.

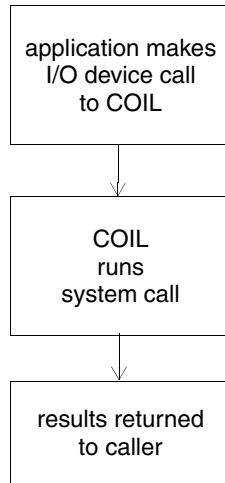
Figure 9-5 shows the flow of control for a vThreads application running I/O with worker tasks. Note that the application is unaware that worker tasks are used.

Figure 9-5 **Non-blocking vThreads I/O (Worker Tasks Present)**



In contrast, [Figure 9-6](#) shows the flow of control for a COIL-based application running I/O without worker tasks.

Figure 9-6 **Blocking COIL I/O (No Worker Tasks)**



It is the responsibility of the application to check whether **COIL_SYSCALL_PENDING** was returned from the system call and to decide what to do until the **COIL_EVENT_SYSCALL_COMPLETE** pseudo-interrupt arrives.

Non-blocking COIL I/O (Worker Tasks Present)

When worker tasks are present, the core OS might defer device I/O calls. In this case, the I/O system call returns immediately with the **COIL_SYSCALL_PENDING** return code. Once the device I/O operation completes, the core OS delivers a **COIL_EVENT_SYSCALL_COMPLETE** pseudo-interrupt to the COIL-based partition OS, which in turn passes it to the application's pseudo-interrupt handler.

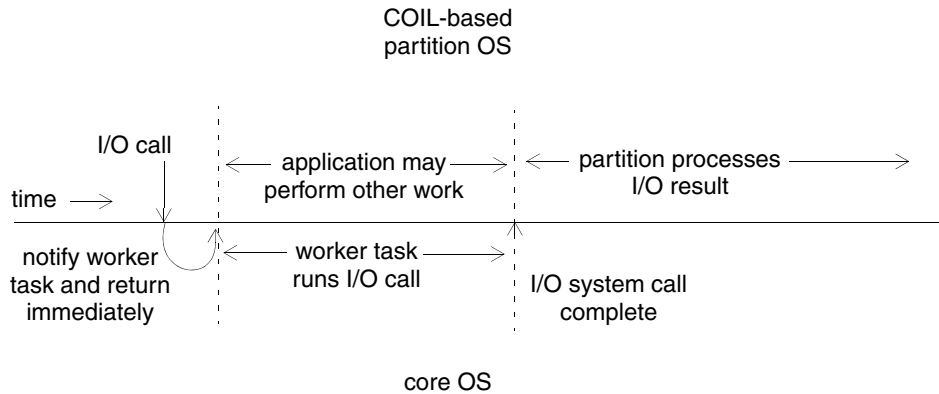
To correlate the original device I/O call with the subsequent **COIL_EVENT_SYSCALL_COMPLETE** pseudo-interrupt, the application passes two user-provided, unique IDs to each device I/O API routine; the IDs are returned in the corresponding pseudo-interrupt structure. It is the responsibility of the user partition OS to manage the unique IDs and to perform the correlation. The pseudo-interrupt structure is defined as follows, where the first two data fields

(**data1** and **data2**) correspond to the IDs if **evtType** is **COIL_EVENT_SYSCALL_COMPLETE**:

```
typedef struct COIL_EVENT
{
    COIL_EVENT_TYPE  evtType;      /* event type */
    int              data1;        /* event data word 1 */
    int              data2;        /* event data word 2 */
    int              data3;        /* event data word 3 */
    int              data4;        /* event data word 4 */
} coil_event;
```

Figure 9-7 shows an overall flow of a COIL I/O call when worker tasks are present (non-blocking I/O).

Figure 9-7 Non-blocking COIL I/O (Worker Tasks Present)



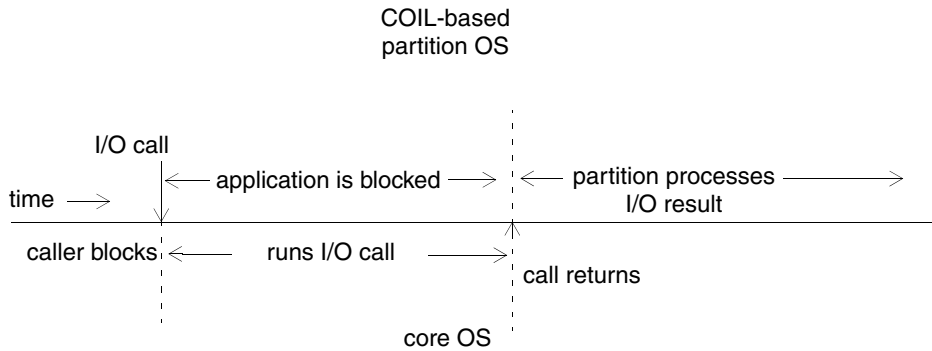
Blocking I/O (No Worker Tasks)

In the case where worker tasks are not used, the device I/O routines may block in the core OS. In this case, the COIL I/O system call does not return to the application until the operation has been completed.

When worker tasks are absent, the unique IDs provided in the device I/O APIs are ignored.

Figure 9-8 shows an overall flow of a COIL I/O call when worker tasks are not present.

Figure 9-8 **Blocking COIL I/O (No Worker Tasks)**



A

VxWorks 5.5

A.1 Introduction	269
A.2 VxWorks Tasks	270
A.3 Intertask Communications	294
A.4 VxWorks Events	317
A.5 Watchdog Timers	322
A.6 Interrupt Service Routines	323

A.1 Introduction



NOTE: The vThreads partition OS is based on the VxWorks 5.5 RTOS. The basics of VxWorks 5.5 are described in this appendix, which is taken almost verbatim from the *VxWorks Programmer's Guide*, 5.5. In this appendix, VxWorks refers to VxWorks 5.5. In addition, what this appendix calls tasks are called threads in a vThreads partition. The scheduler that is mentioned is equivalent to the one that schedules threads in a partition.

For information on how vThreads differs from VxWorks 5.5, see [2. Developing vThreads Applications](#).

Real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment lets a real-time

application be constructed as a set of independent tasks, each with its own thread of execution and set of system resources. The intertask communication facilities let these tasks synchronize and communicate in order to coordinate their activity. In VxWorks, the intertask communication facilities range from fast semaphores to message queues and from pipes to network-transparent sockets.

Another key run-time facility is hardware interrupt handling, because interrupts are the usual mechanism to inform a system of external events.

This appendix uses the qualifier *Wind* to identify certain VxWorks kernel objects.

A.2 VxWorks Tasks

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

The POSIX standard includes the concept of a thread, which is similar to a task, but with some additional features. For details, see [5.4 POSIX Threads](#), p.92.

A.2.1 Multitasking

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

A task's context includes:

- a thread of execution (that is, the task's program counter)
- the CPU registers and (optionally) floating-point registers
- a stack for dynamic variables and routine calls

- I/O assignments for standard input, output, and error
- a delay timer
- a time-slice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

A.2.2 Task State Transition

The kernel maintains the current state of each task in the system. A task changes from one state to another as a result of kernel routine 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 lets a task be created and activated with a single routine. Tasks can be deleted from any state.

Table A-1 **Task State Symbols**

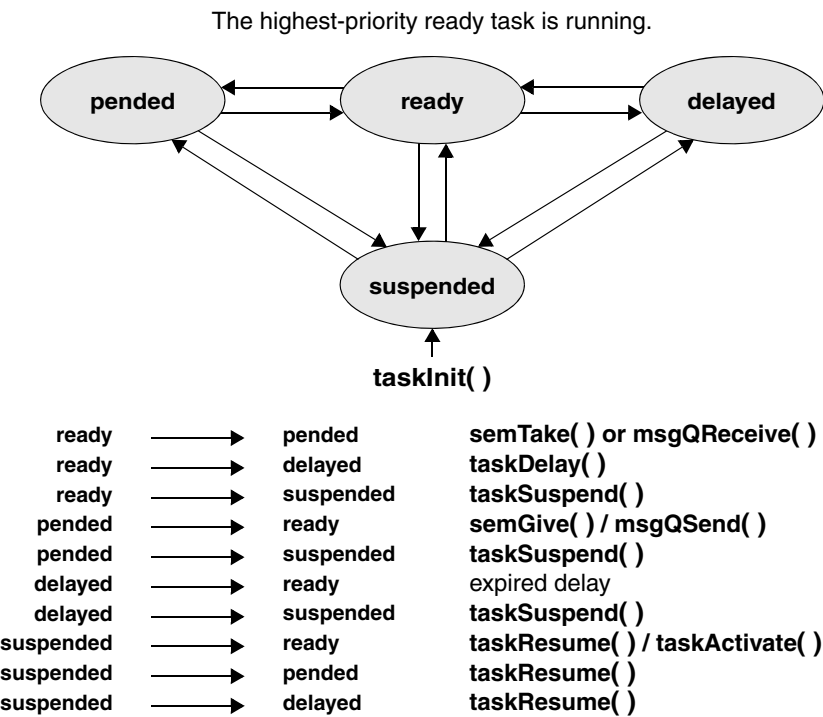
State Symbol	Description
READY	The state of a task that is not waiting for any resource other than the CPU.
PEND	The state of a task that is blocked due to the unavailability of some resource.
DELAY	The state of a task that is asleep for some duration.
SUSPEND	The state of a task that is unavailable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

Table A-1 Task State Symbols (cont'd)

State Symbol	Description
PEND + S + T	The state of a task that is both pended with a timeout value and suspended.
state + I	The state of task specified by state, plus an inherited priority.

Table A-1 describes the state symbols that you see when working with the development tools. Figure A-1 shows the corresponding state diagram of the Wind kernel states.

Figure A-1 Task State Transitions



A.2.3 Wind Task Scheduling

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

The routines that control task scheduling are listed in [Table A-2](#).

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

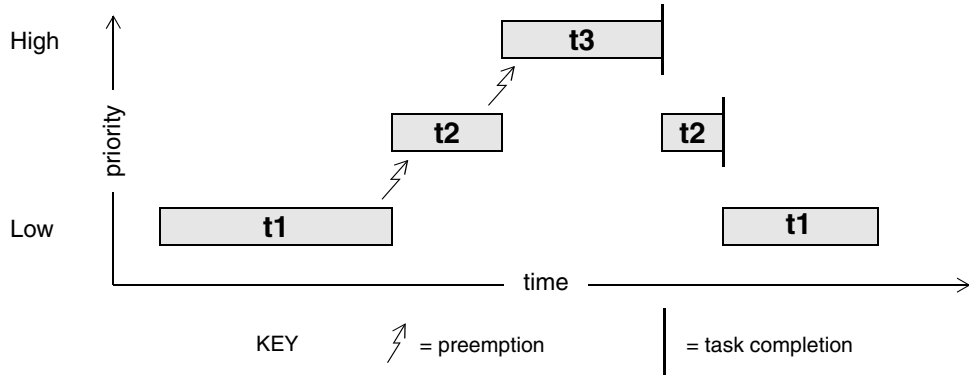
Call	Description
kernelTimeSlice()	Controls round-robin scheduling.
taskPrioritySet()	Changes the priority of a task.
taskLock()	Disables task rescheduling.
taskUnlock()	Enables task rescheduling.

POSIX also provides a scheduling interface. For more information, see [5.5 POSIX Scheduling Interface](#), p.97.

Priority-Preemptive Scheduling

A priority-preemptive scheduler preempts the CPU when a task has a higher priority than the running task. 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 A-2](#), task **t1** is preempted by higher-priority task **t2**, which in turn is preempted by **t3**. When **t3** completes, **t2** continues running. When **t2** completes running, **t1** continues running.

Figure A-2 Priority Preemption



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

Round-Robin Scheduling

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

Round-robin scheduling is enabled by calling **kernelTimeSlice()**, which takes a parameter for a time slice, or interval. This interval is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task. Thus, the tasks rotate, each running for an equal interval of time. No task gets a second slice of time before all other tasks in the priority group have been allowed to run.

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

If round-robin scheduling is enabled, and preemption is enabled for the running 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.

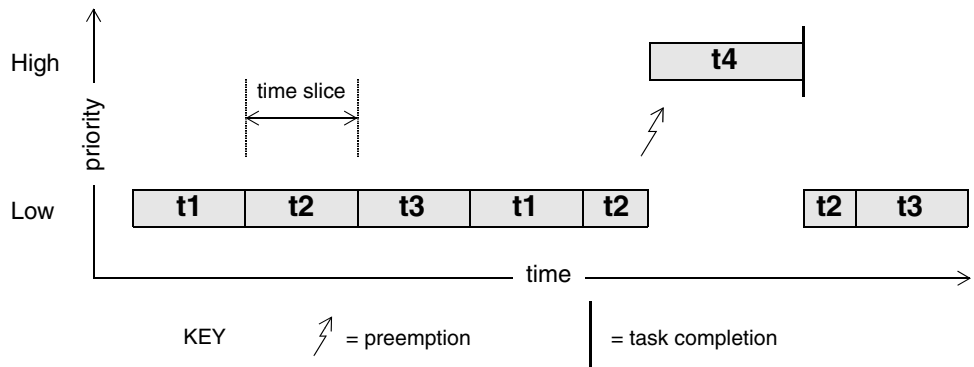
Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.

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 to run. In the case of preemption, the task resumes running 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 running task is not incremented.

Time-slice counts are accrued by the task that is running when a system tick occurs, regardless of whether the task has run 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 run for either more or less total CPU time than its allotted time slice.

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

Figure A-3 **Round-Robin Scheduling**



Preemption Locks

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

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

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

Preemption locks can be used to achieve mutual exclusion; however, keep the duration of preemption locking to a minimum. For more information, see [A.3.2 Mutual Exclusion](#), p.295.

For information on possible interaction with health monitoring, see [Dispatching Rules](#), p.205.

A Comparison of **taskLock()** and **intLock()**

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

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

Driver Support Task Priority

All application tasks should be priority 100 - 250. However, driver support tasks (tasks associated with an ISR) can be in the range of 51-99. These tasks are crucial. For example, if a support task fails while copying data from a chip, the device loses that data (for example, a network interface, an HDLC, and so on). The system **netTask()** is at priority 50, so user tasks should not be assigned priorities below that task. If they are, the network connection could die and prevent debugging capabilities with Workbench.

A.2.4 Task Control

The following sections give an overview of the basic VxWorks task routines, which are found in the VxWorks library **taskLib**. These routines provide the means for task creation and control, as well as for retrieving information about tasks. See the *VxWorks 653 vThreads API Reference* entry for **taskLib** for further information.

For interactive use, you can control VxWorks tasks from the host or target shell. See the *Workbench User's Guide, VxWorks 653 Version*.

Task Creation and Activation

The routines listed in [Table A-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 ten arguments to be passed to the main routine as startup parameters:

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

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

Table A-3 **Task Creation Routines**

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

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

Task Stack

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

Task Names and IDs

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

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

You may not want to name some or all your application's tasks. If a **NULL** pointer is supplied for the *name* argument of `taskSpawn()`, 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 A-4](#) manage task IDs and names.

Table A-4 Task Name and ID Routines

Call	Description
<code>taskName()</code>	Gets the task name associated with a task ID.
<code>taskNameToId()</code>	Looks up the task ID associated with a task name.
<code>taskIdSelf()</code>	Gets the calling task's ID.
<code>taskIdVerify()</code>	Verifies the existence of a specified task.

Task Options

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

Table A-5 **Task Options**

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

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

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

For example:

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

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

After a task is spawned, you can examine or alter task options by using the routines listed in [Table A-6](#). Only the **VX_UNBREAKABLE** option can be altered.

Table A-6 Task Option Routines

Call	Description
<code>taskOptionsGet()</code>	Examines task options.
<code>taskOptionsSet()</code>	Sets task options.

Task Information

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

Table A-7 Task Information Routines

Call	Description
<code>taskIdListGet()</code>	Fills an array with the IDs of all active tasks.
<code>taskInfoGet()</code>	Gets information about a task.
<code>taskPriorityGet()</code>	Examines the priority of a task.
<code>taskRegsGet()</code>	Examines a task's registers (cannot be used with the current task).
<code>taskRegsSet()</code>	Sets a task's registers (cannot be used with the current task).
<code>taskIsSuspended()</code>	Checks whether a task is suspended.
<code>taskIsReady()</code>	Checks whether a task is ready to run.
<code>taskTcb()</code>	Gets a pointer to a task's control block.

Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in [Table A-8](#) to delete tasks and to protect tasks from unexpected deletion.

Table A-8 **Task-Deletion Routines**

Call	Description
exit()	Terminates the calling task and frees memory (task stacks and task control blocks only) Memory that a task allocates while it runs is not freed when the task is terminated.
taskDelete()	Terminates a specified task and frees memory (task stacks and task control blocks only).*
taskSafe()	Protects the calling task from deletion.
taskUnsafe()	Undoes a taskSafe() (makes the calling task available for deletion).



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

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

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



NOTE: You can use VxWorks events to send an event when a task finishes running. For more information, see [A.4 VxWorks Events](#), p.317.

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

Using **taskSafe()** to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with **taskSafe()** is blocked. When finished with its critical resource, the protected task can make itself available for deletion by calling **taskUnsafe()**, which readies any deleting task. To support

nested deletion-safe regions, a count is kept of the number of times **taskSafe()** and **taskUnsafe()** are called. Deletion is allowed only when the count is zero, that is, there are as many “unsafes” as “safes.” Only the calling task is protected. A task cannot make another task safe or unsafe from deletion.

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

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

Deletion safety is often coupled closely with mutual exclusion, as in this example. For convenience and efficiency, a special kind of semaphore, the mutual-exclusion semaphore, offers an option for deletion safety. For more information, see [Mutual-Exclusion Semaphores](#), p.302.

Task Control

The routines listed in [Table A-9](#) provide direct control over a task’s running.

Table A-9 Task Control Routines

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

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

While they run, tasks may need to be restarted 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 **sysClkRateGet()** routine returns the speed of the system clock in ticks per second. Instead of **taskDelay()**, you can use the POSIX **nanosleep()** routine to specify a delay directly in time units. Only the units are different. The resolution of both delay routines is the same, and depends on the system clock. For details, see [5.2 POSIX Clocks and Timers](#), p.90.

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 possible only with **taskDelay()**. The **nanosleep()** routine considers it an error.

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

A.2.5 Tasking Extensions

To let additional task-related facilities be added to the system, VxWorks provides hook routines that let additional routines be called when 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 A-10](#). For more information, see the reference entry for **taskHookLib**.

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

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

Table A-10 Task Create, Switch, and Delete Hooks (cont'd)

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

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

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

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

User-installed switch hooks are called within the kernel context and therefore do not have access to all VxWorks facilities. [Table A-11](#) summarizes the routines that can be called from a task switch hook. In general, any routine that does not involve the kernel can be called.

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

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

For information about POSIX extensions, see [5. Developing POSIX Applications](#).

A.2.6 Task Error Status: **errno**

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

Layered Definitions of **errno**

In VxWorks, **errno** is simultaneously defined in two different ways. There is, as in ANSI C, an underlying global variable called **errno**, which you can display by

name using the development tools. (See the *Workbench User's Guide, VxWorks 653 Version*.) However, **errno** is also defined as a macro in **errno.h**. This is the definition visible to all VxWorks, except one routine. The macro is defined as a call to an **__errno()** routine that returns the address of the global **errno** variable. The **__errno()** routine is the one routine that does not itself use the macro definition for **errno**). This yields a useful feature: because **__errno()** is a routine, you can place breakpoints on it to determine where a particular error occurs.

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

```
errno = someErrorNumber;
```

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

A Separate **errno** Value for Each Task

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

This is accomplished by saving and restoring **errno** on the interrupt stack as part of the interrupt enter and exit code. Thus, regardless of the VxWorks context, an error code can be stored or consulted with direct manipulation of the global variable **errno**.

Error Return Convention

Almost all VxWorks routines follow a convention that indicates simple success or failure of their operation by the actual return value of the routine. Many routines return only the status values **OK** (0) or **ERROR** (-1). Some routines that normally return a nonnegative number (for example, **open()** returns a file descriptor) also return **ERROR** to indicate an error. Routines that return a pointer usually return **NULL** (0) to indicate an error. In most cases, a routine 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 routine gets an error indication from a call to another routine, it usually returns its own error indication without modifying **errno**. Thus, the value of **errno** that is set in the lower-level routine remains available as the indication of error type.

For example, if **malloc()** fails because insufficient memory remains in the pool, it sets **errno** to a code indicating an insufficient-memory error was encountered in the memory allocation library, **memLib**. The **malloc()** routine then returns **NULL** to indicate the failure. The calling routine, receiving the **NULL** from **malloc()**, then returns its own error indication of **ERROR**. However, it does not alter **errno** leaving it at the “insufficient memory” code set by **malloc()**. For example:

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

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

Assignment of Error Status Values

A VxWorks **errno** value encodes the module (library) that issues the error in the most significant two bytes. It uses the least significant two bytes for individual error numbers. 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 left-shifted 16, and all negative values) are available for application use.

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

A.2.7 Task Exception Handling

Errors in program code or data can cause hardware exception conditions such as illegal instructions, bus or address errors, divide by zero, and so forth. The VxWorks exception handling package takes care of all such exceptions. The default exception handler suspends the task that caused the exception, and saves the state of the task at the point of the exception. The kernel and other tasks continue uninterrupted. A description of the exception is transmitted to the development

tools, which can be used to examine the suspended task. For details, see the *Workbench User's Guide, VxWorks 653 Version*.

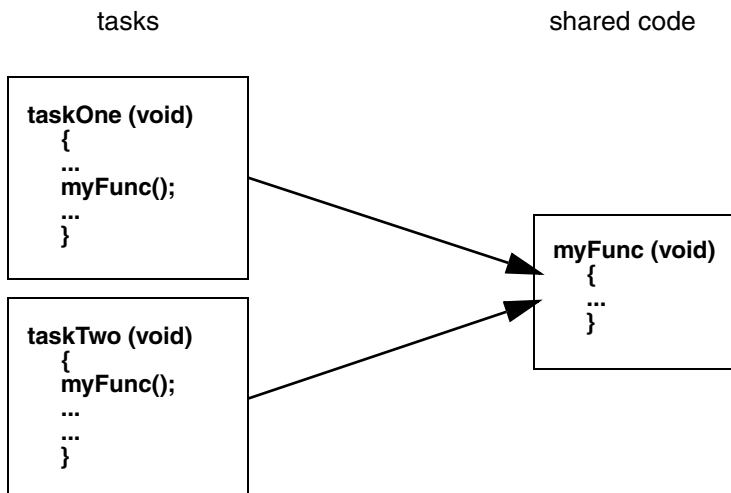
Tasks can also attach their own handlers for certain hardware exceptions through the signal facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. A user-defined signal handler is useful for recovering from catastrophic events. Typically, **setjmp()** is called to define the point in the program where control is restored, and **longjmp()** is called in the signal handler to restore that context. The **longjmp()** routine 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 [A.3.6 Signals](#), p.315 and in the reference entry for **sigLib**.

A.2.8 Shared Code and Reentrancy

In VxWorks, it is common for a single copy of a routine or routine library to be called by many different tasks. For example, many tasks may call **printf()**, but there is only a single copy of the routine in the system. A single copy of code that is run 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 A-4](#).

Figure A-4 Shared Code



Shared code must be reentrant. A routine 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 routine 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 **ldiv_r()** routine, you can assume that **ldiv()** is not reentrant.

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

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

The majority of VxWorks routines use the following reentrancy techniques:

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

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

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



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

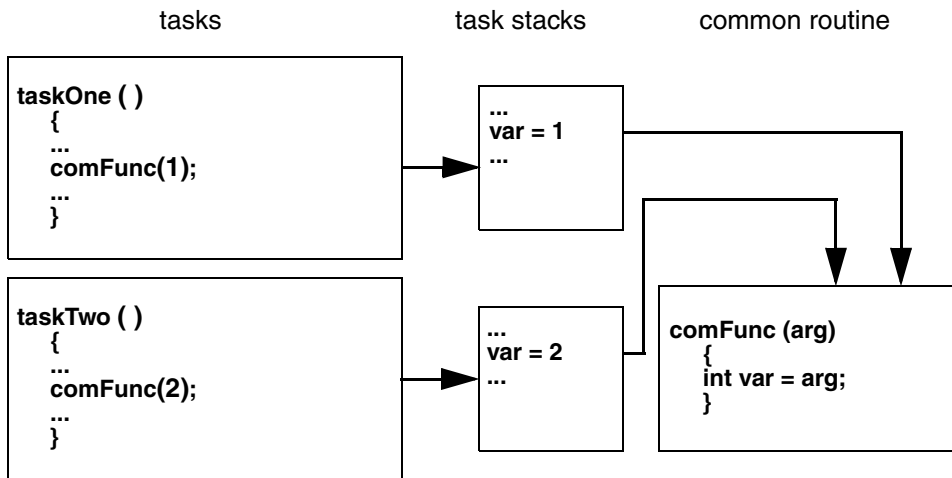
Dynamic Stack Variables

Many routines 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 routine call.

Routines 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 A-5](#).

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



Guarded Global and Static Variables

Some libraries encapsulate access to common data. This kind of library requires some caution because the routines are not inherently reentrant. Multiple tasks simultaneously calling 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 running critical sections of code. The usual mutual-exclusion mechanism is the mutex semaphore facility provided by **semMLib** and described in [Mutual-Exclusion Semaphores](#), p.302.

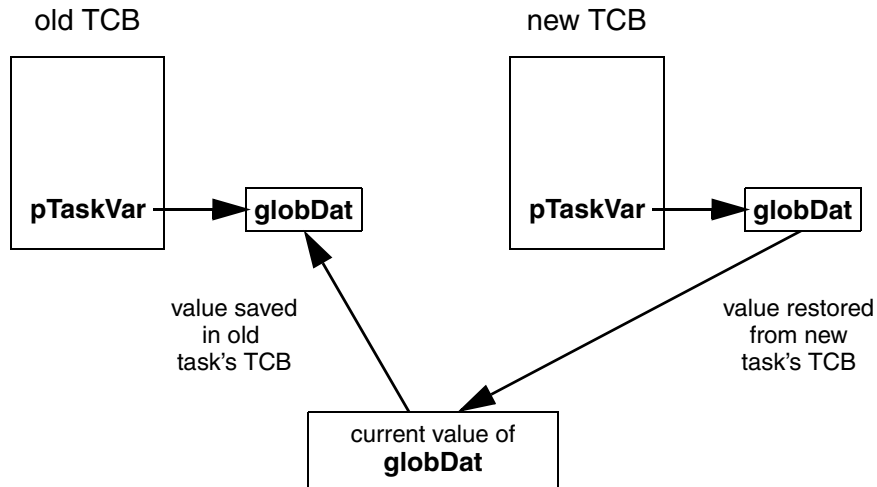
Task Variables

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 lets four-byte variables 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 (four-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 A-6](#). This facility is provided by **taskVarAdd()**, **taskVarDelete()**, **taskVarSet()**, and **taskVarGet()**, which are described in the reference entry for **taskVarLib**.

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



Use this mechanism sparingly. Each task variable adds a few microseconds to the context switching time for its task, because the value of the variable must be saved and restored as part of the task's context. Consider collecting all a module's (library'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 (library).

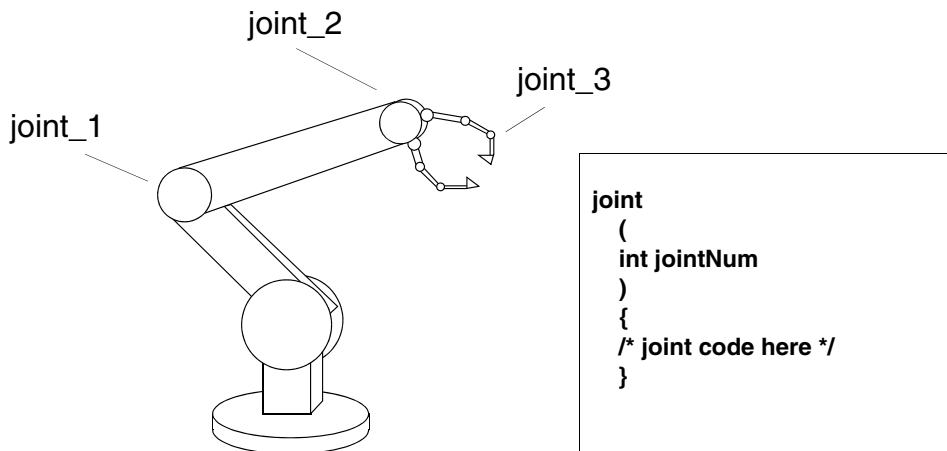
Multiple Tasks with the Same Main Routine

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.290 apply to the entire task.

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

In [Figure A-7](#), multiple joints of the mechanical arm use the same code. The tasks manipulating the joints call `joint()`. The joint number (`jointNum`) is used to indicate which joint on the arm to manipulate.

Figure A-7 Multiple Tasks Using the Same Code



A.2.9 VxWorks System Tasks

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

Root Task: tUsrRoot

The root task is the first task that the kernel runs. The entry point of the root task is **usrRoot()** in:

installDir/target/config/all/usrConfig.c

and initializes most VxWorks facilities. It spawns such tasks as the logging task, the exception task, the network task, and the **tRlogind** daemon. Normally, the root task terminates and is deleted after all initialization has occurred.

Logging Task: tLogTask

The log task, **tLogTask**, is used by VxWorks modules (libraries) to log system messages without having to perform I/O in the current task context. For more information, see the reference entry for **logLib**.

Exception Task: tExcTask

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

Tasks for Optional Components

The following VxWorks system tasks are created if their associated configuration constants are defined.

tShell

If you have included the target shell in the VxWorks configuration, it is spawned as this task. Any routine or task that is called from the target shell, rather than spawned, runs in the **tShell** context. Configure VxWorks with the **INCLUDE_SHELL** component to include the target shell.

tTelnetd

If you have included the target shell and the **telnet** facility in the VxWorks configuration, this daemon lets remote users log in to VxWorks with **telnet**. It accepts a remote login request from another VxWorks or host system and spawns the input task **tTelnetInTask** and output task **tTelnetOutTask**. These

tasks exist as long as the remote user is logged on. During the remote session, the shell's (and any other task's) input and output are redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**. See the reference entry for **ptyDrv** for further explanation. Configure VxWorks with the **INCLUDE_TELNET** component to include the telnet facility.

A.3 Intertask Communications

The complement to the multitasking routines described in [A.2 VxWorks Tasks](#), p.270 is the intertask communication facilities. These facilities permit independent tasks to coordinate their actions.

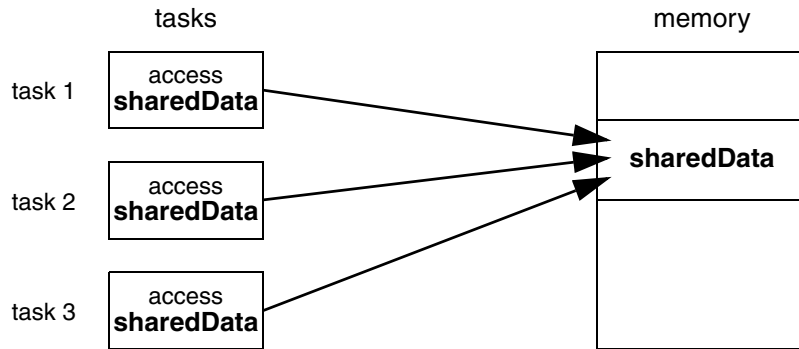
VxWorks supplies a rich set of intertask communication mechanisms, including:

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

A.3.1 Shared Data Structures

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

Figure A-8 **Shared Data Structures**



A

A.3.2 Mutual Exclusion

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

For information about POSIX mutexes, see [5.7 POSIX Mutexes and Condition Variables](#), p.108.

Interrupt Locks and Latency

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

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

While this solves problems involving mutual exclusion with ISRs, it is inappropriate as a general-purpose mutual-exclusion method for most real-time systems, because it prevents the system from responding to external events for the duration of these locks. Interrupt latency is unacceptable whenever an immediate

response to an external event is required. However, interrupt locking can sometimes be necessary where mutual exclusion involves ISRs. In any situation, keep the duration of interrupt lockouts short.



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

Preemptive Locks and Latency

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

```
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 run until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region. While this kind of mutual exclusion is simple, if you use it, make sure to keep the duration short. A better mechanism is provided by semaphores, discussed in [A.3.3 Semaphores](#), p.296.



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

A.3.3 Semaphores

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

- For mutual exclusion semaphores interlock access to shared resources. They provide mutual exclusion with finer granularity than either interrupt disabling or preemptive locks, discussed in [A.3.2 Mutual Exclusion](#), p.295.

- For synchronization semaphores coordinate a task's running with external events.

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

- **Binary**

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

- **Mutual exclusion**

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

- **Counting**

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

VxWorks provides not only the Wind semaphores, designed expressly for VxWorks, but also POSIX semaphores, designed for portability. An alternate semaphore library provides the POSIX-compatible semaphore interface. See [5.6 POSIX Semaphores](#), p.101.

The semaphores described here are for use on a single CPU.

Semaphore Control

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

Table A-12 **Semaphore Control Routines**

Call	Description
semBCreate()	Allocates and initializes a binary semaphore.
semCCreate()	Allocates and initializes a counting semaphore.
semDelete()	Terminates and frees a semaphore.
semFlush()	Unblocks all tasks that are waiting for a semaphore.

Table A-12 Semaphore Control Routines (cont'd)

Call	Description
semGive()	Gives a semaphore.
semMCreate()	Allocates and initializes a mutual-exclusion semaphore.
semTake()	Takes a semaphore.

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



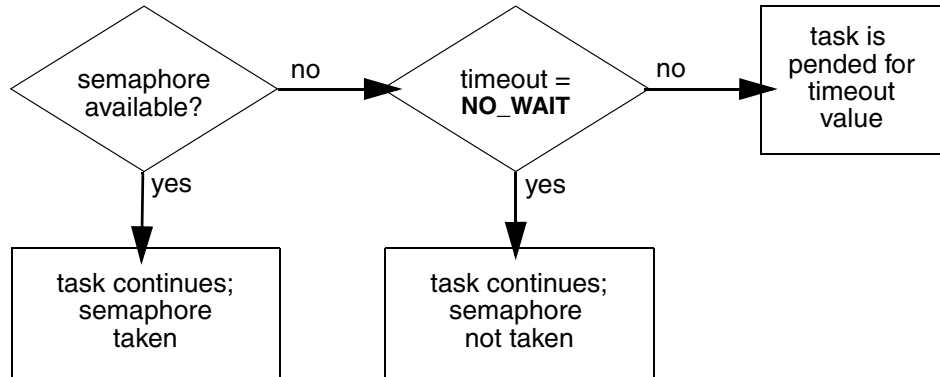
WARNING: The **semDelete()** routine terminates a semaphore and deallocates all associated memory. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore that another task still requires. Do not delete a semaphore unless the same task first succeeds 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.302 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.

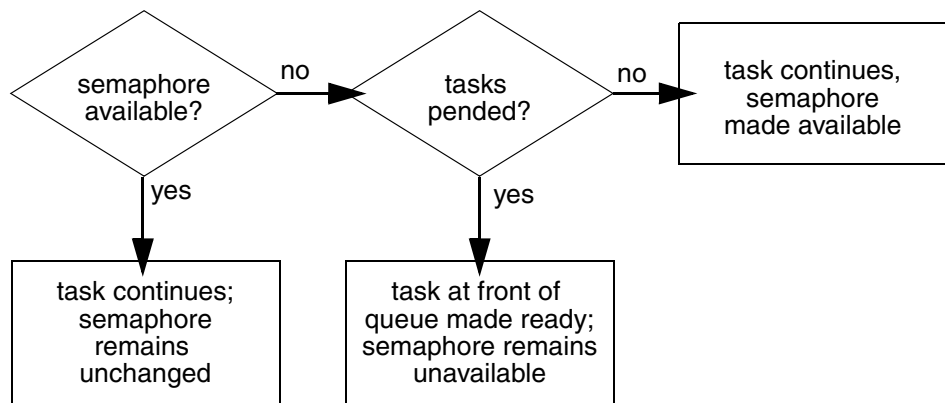
A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with **semTake()**, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call. See [Figure A-9](#). If the semaphore is available (full), the semaphore becomes unavailable (empty) and the task continues running 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 A-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 A-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, the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).

Figure A-10 **Giving a Semaphore**



Mutual Exclusion

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

```
/* includes */
#include "vxWorks.h"
#include "semLib.h"

SEM_ID semMutex;

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

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

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

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

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

Synchronization

When used for task synchronization, a semaphore can represent a condition or event that a task is waiting for. Initially, the semaphore is unavailable (empty). A task or ISR signals the occurrence of the event by giving the semaphore (see [A.6 Interrupt Service Routines](#), p.323 for a complete discussion of ISRs). Another task waits for the semaphore by calling **semTake()**. The waiting task blocks until the event occurs and the semaphore is given.

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

In [Example A-1](#), **init()** creates the binary semaphore, attaches an ISR to an event, and spawns a task to process the event. The **task1()** routine runs until it calls

semTake(). It remains blocked until an event causes the ISR to call **semGive()**. When the ISR completes, **task1()** runs to process the event. There is an advantage of handling event processing within the context of a dedicated task: less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

Example A-1 Using Semaphores for Task Synchronization

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

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

SEM_ID syncSem;          /* ID of sync semaphore */

init (
    int someIntNum
)
{

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

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

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

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

Broadcast synchronization lets 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 **semFlush()** routine addresses this class of synchronization problem by unblocking all tasks pended on a semaphore.

Mutual-Exclusion Semaphores

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

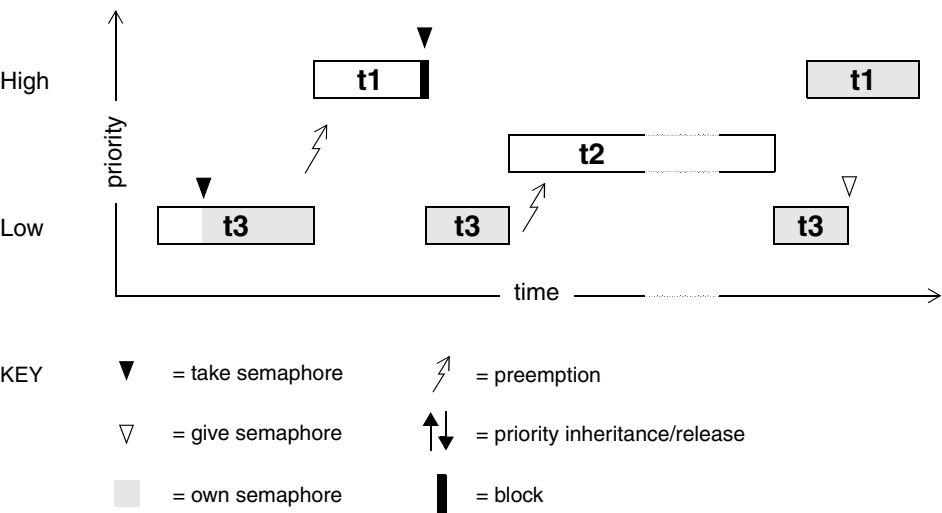
The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

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

Priority Inversion

Figure A-11 illustrates a situation called priority inversion.

Figure A-11 Priority Inversion

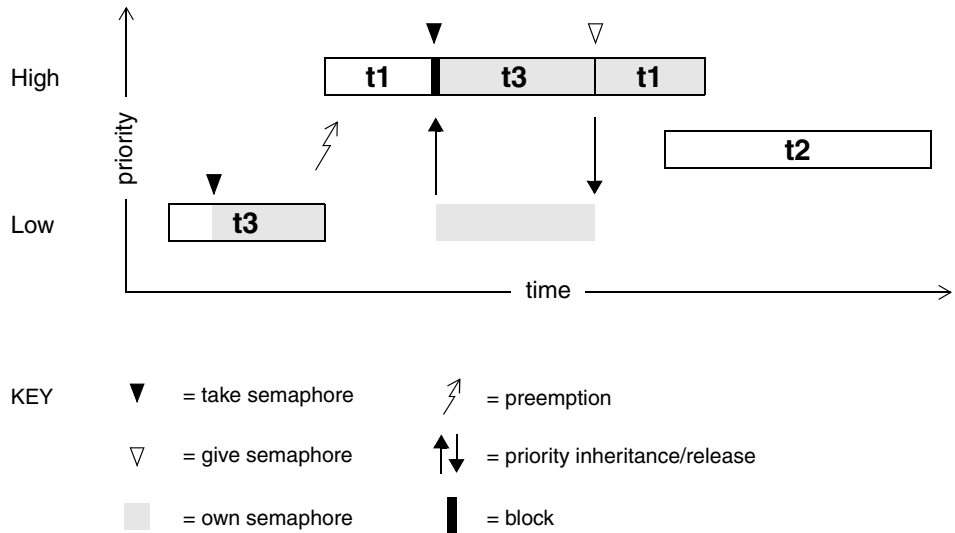


Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for a lower-priority task to complete. Consider the scenario in Figure A-11: **t1**, **t2**, and **t3** are tasks of high, medium, and low priority, respectively. **t3** has acquired some resource by taking its associated binary guard semaphore.

When **t1** preempts **t3** and contends for the resource by taking the same semaphore, it becomes blocked. If you can be assured that **t1** will block no longer than the time it normally takes **t3** to finish with the resource, there is no problem, because the resource cannot be preempted. However, the low-priority task is vulnerable to preemption by medium-priority tasks (like **t2**), which could inhibit **t3** from relinquishing the resource. This condition could persist, blocking **t1** for an indefinite period of time.

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

Figure A-12 **Priority Inheritance**



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

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

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

Deletion Safety

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

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

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

Recursive Resource Access

Mutual-exclusion semaphores can be taken recursively. This means that the semaphore can be taken more than once by the task that holds it before finally being released. Recursion is useful for a set of routines that must call each other but that also require mutually exclusive access to a resource. This is possible because the system keeps track of which task is holding 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 A-2 Recursive Use of a Mutual-Exclusion Semaphore

```
/* Function A requires access to a resource which it acquires by taking
 * mySem;
 * Function A may also need to call function B, which also requires mySem;
 */

/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;

/* Create a mutual-exclusion semaphore. */
init ()
```

```

    {
        mySem = semMCreate (SEM_Q_PRIORITY);
    }
funcA ()
{
    semTake (mySem, WAIT_FOREVER);
    printf ("funcA: Got mutual-exclusion semaphore\n");
    ...
    funcB ();
    ...

    semGive (mySem);
    printf ("funcA: Released mutual-exclusion semaphore\n");
}
funcB ()
{
    semTake (mySem, WAIT_FOREVER);
    printf ("funcB: Got mutual-exclusion semaphore\n");
    ...
    semGive (mySem);
    printf ("funcB: Releases mutual-exclusion semaphore\n");
}

```

A

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, the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking. [Table A-13](#) shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of three (3).

Table A-13 **Counting Semaphore Example**

Semaphore Call	Count after Call	Resulting Behavior
semCCreate()	3	Semaphore initialized with an initial count of 3.
semTake()	2	Semaphore taken.
semTake()	1	Semaphore taken.
semTake()	0	Semaphore taken.

Table A-13 **Counting Semaphore Example** (cont'd)

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

Counting semaphores are useful for guarding multiple copies of resources. For example, the use of five tape drives might be coordinated using a counting semaphore with an initial count of five, 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 **semCCreate()**.

Special Semaphore Options

The uniform Wind semaphore interface includes two special options. These options are not available for the POSIX-compatible semaphores described in [5.6 POSIX Semaphores](#), p. 101.

Timeouts

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

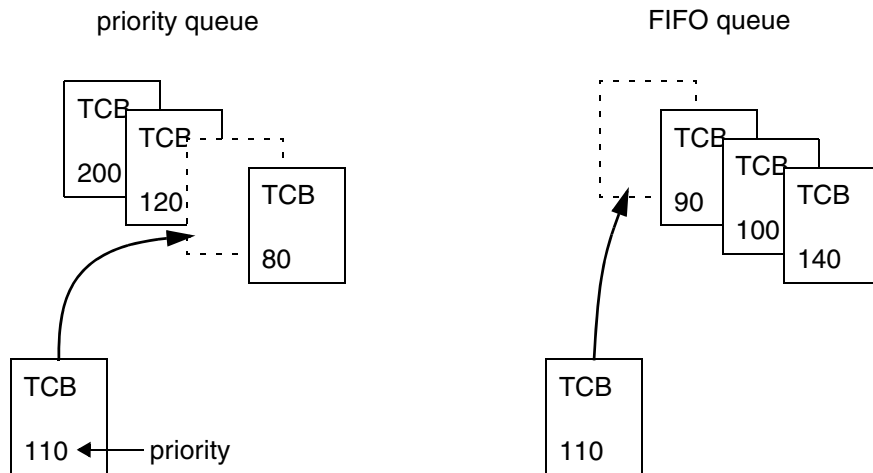
This behavior is controlled by a parameter to **semTake()** that specifies the amount of time in ticks that the task is willing to wait in the pended state. If the task succeeds in taking the semaphore within the allotted time, **semTake()** returns **OK**. The **errno** set when a **semTake()** returns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

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

Queues

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

Figure A-13 **Task Queue Types**



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

Semaphores and VxWorks Events

This section describes using VxWorks events with semaphores. You can also use VxWorks events with other VxWorks objects. For more information, see [A.4 VxWorks Events](#), p.317.

Using Events

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

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

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

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

Existing VxWorks API

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

Performance Impact

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

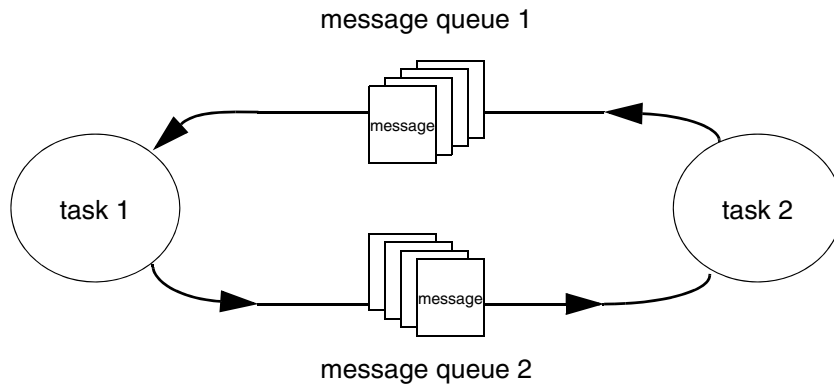
The **semDestroy()** routine performance is impacted in cases where a task is waiting for events from the semaphore, since the task has to be awakened. Also note that, in this case, events need not be sent.

A.3.4 Message Queues

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

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

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



Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction. See [Figure A-14](#).

There are two message-queue libraries in VxWorks. The first of these, **msgQLib**, provides Wind message queues, designed expressly for VxWorks. The second, **mqPxBLib**, is compatible with the POSIX standard (1003.1b) for real-time

extensions. See [5.5.1 Comparison of POSIX and Wind Scheduling](#), p.98 for a discussion of the differences between the two message-queue designs.

Wind Message Queues

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

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

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

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

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

A task receives a message from a message queue with **msgQReceive()**. If messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, 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()** routine lets 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 A-3 Wind Message Queues

```

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

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

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

MSG_Q_ID myMsgQId;

task2 (void)
{
    char msgBuf[MAX_MSG_LEN];

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

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

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

```

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

Displaying Message Queue Attributes

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

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

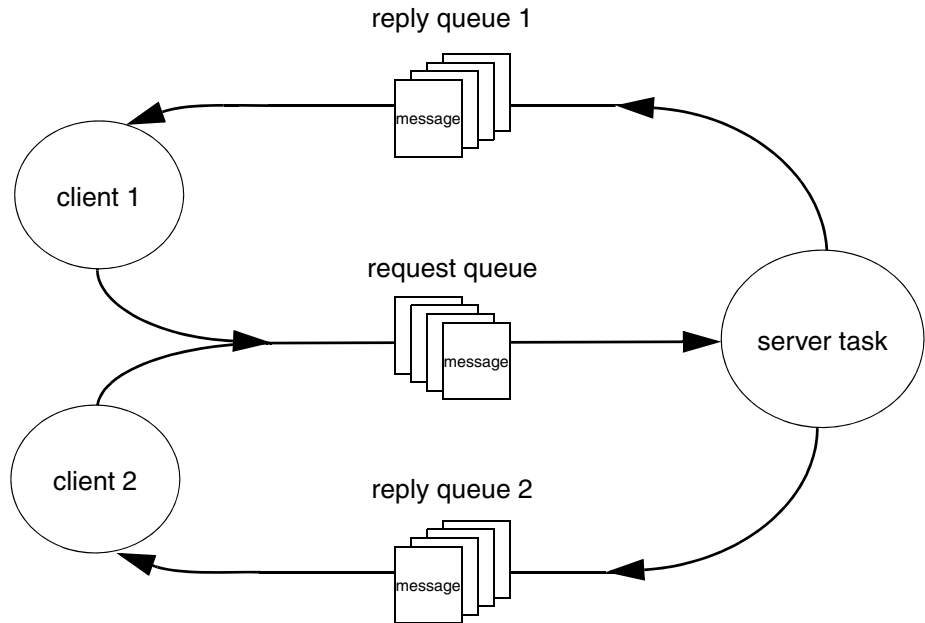
Servers and Clients with Message Queues

Real-time systems are often structured using a client-server model of tasks. In this model, server tasks accept requests from client tasks to perform some service, and usually return a reply. The requests and replies are usually made in the form of intertask messages. In VxWorks, message queues or pipes (see [A.3.5 Pipes](#), p.315) are a natural way to implement this.

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

The same architecture can be achieved with pipes instead of message queues, or by other means that are tailored to the needs of the particular application.

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



Message Queues and VxWorks Events

This section describes using VxWorks events with message queues. You can also use VxWorks events with other VxWorks objects. For more information, see [A.4 VxWorks Events](#), p.317.

Using Events

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

Only one task can be registered with a message queue at any given time. The events a message queue sends to a task can be retrieved by the task using routines

in **eventLib**. Details on when message queues send events are documented in the reference entry for **msgQEvStart()**.

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

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

Existing VxWorks API

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

Performance Impact

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

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

A.3.5 Pipes

Pipes provide an alternative interface to the message queue facility that goes through the VxWorks I/O system. Pipes are virtual I/O devices managed by the driver **pipeDrv**. The **pipeDevCreate()** routine 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 call *ioctl* routines. As they do with other I/O devices, tasks block when they read from an empty pipe until data is available, and block when they write to a full pipe until there is space available. Like message queues, ISRs can write to a pipe, but cannot read from a pipe.

As I/O devices, pipes provide one important feature that message queues cannot—the ability to be used with **select()**. This routine lets a task 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.

Pipes let you implement a client-server model of intertask communications. See *Servers and Clients with Message Queues*, p.312.

A.3.6 Signals

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

Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism. In general, signal handlers should be treated like ISRs. No routine should be called from a signal handler that might cause the handler to block. Because signals are asynchronous, it is difficult to predict which resources might be unavailable when a particular signal is raised. To be perfectly safe, call only those routines that can safely be called from an ISR

(see [Table A-19](#)). Deviate from this practice only when you are sure your signal handler cannot create a deadlock situation.

The Wind kernel supports two types of signal interface: UNIX BSD-style signals and POSIX-compatible signals. The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b. For more information, see [5.9 POSIX Queued Signals](#), p. 120. For the sake of simplicity, Wind River recommends that you use only one interface type in a given application, rather than mixing routines from different interfaces.

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



NOTE: The VxWorks implementation of **sigLib** does not impose any special restrictions on operations on **SIGKILL**, **SIGCONT**, and **SIGSTOP** signals such as those imposed by UNIX. For example, the UNIX implementation of **signal()** cannot be called on **SIGKILL** and **SIGSTOP**.

Basic Signal Routines

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

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

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

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

Signal Configuration

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

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

POSIX 1003.1b Compatible Call	UNIX BSD Compatible Call	Description
<code>kill()</code>	<code>kill()</code>	Sends a signal to a task.
<code>raise()</code>	N/A	Sends a signal to yourself.
<code>sigaction()</code>	<code>sigvec()</code>	Examines or sets the signal handler for a signal.
<code>sigemptyset()</code> <code>sigfillset()</code> <code>sigaddset()</code> <code>sigdelset()</code> <code>sigismember()</code>	<code>sigsetmask()</code>	Manipulates a signal mask.
<code>signal()</code>	<code>signal()</code>	Specifies the handler associated with a signal.
<code>sigpending()</code>	N/A	Retrieves a set of pending signals blocked from delivery.
<code>sigprocmask()</code>	<code>sigsetmask()</code>	Sets the mask of blocked signals.
<code>sigprocmask()</code>	<code>sigblock()</code>	Adds to a set of blocked signals.
<code>sigsuspend()</code>	<code>pause()</code>	Suspends a task until a signal is delivered.

A

A.4 VxWorks Events

VxWorks events are included in the standard VxWorks facilities. This section provides a brief summary of VxWorks events. Then, it describes VxWorks events in detail, including their API.



NOTE: This section uses the term events to describe VxWorks events. Do not confuse these references with WindView events.

VxWorks events are a means of communication between tasks and interrupt routines (ISRs), between tasks and other tasks, or between tasks and VxWorks objects. In the context of VxWorks events, these objects are referred to as resources,

and they include semaphores and message queues. Only tasks can receive events; whereas tasks, ISRs, or resources can send them.

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

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

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

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



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

A.4.1 Free Resource Definition

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

- **Mutex Semaphore**

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

- **Binary Semaphore**

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

- **Counting Semaphore**

A counting semaphore is considered free when its count is nonzero and no one is pending on it. Thus, events cannot be used as a mechanism to compute the number of times a semaphore is released or given.

- **Message Queue**

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

A

A.4.2 Single-Task Resource Registration

When a task registers with a resource to send events, it could inadvertently deregister another task that had previously registered with the resource. This prevents the first task from receiving events from the resource with which it registered. Consequently, the task that first registered with the resource could stay in a pend state indefinitely.

VxWorks events provide an option whereby the second task is not allowed to register with the resource if another task is already registered with it. If a second task tries to register with the resource, an error is returned.

A.4.3 Option for Immediate Send

When a task registers with a resource, the default behavior is that the resource does not send VxWorks events to the task immediately, even if it is free at the time of registration. VxWorks events provide an option that lets a task, at the time of registration, request that the resource send the events immediately, if the resource is free at the time of registration.

A.4.4 Option for Automatic Unregister

There are situations in which a task may want to receive events from a resource only once, and then unregister. VxWorks provides an option whereby a registering task can tell the resource to send events only once, and automatically unregister the task when this occurs.

A.4.5 Automatic Unpend upon Resource Deletion

When a resource (a semaphore or message queue) is deleted, **semDelete()** and **msgQDelete()** unpend any task. This prevents the task from pending indefinitely, while waiting for events from the resource being deleted. The pending task then resumes running, and receives an **ERROR** return value from the **eventReceive()** call that caused the task to pend. See also, [Existing VxWorks API](#), p.308 and [Existing VxWorks API](#), p.314.

A.4.6 Task Events Register

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

You do not access the task events register directly. Tasks, ISRs, and resources fill the events register of a particular task by sending events to that task. A task can also send itself events, thereby filling its own events register. Events 25 to 32 (VXEV25 or 0x01000000 to VXEV32 or 0x80000000) are reserved for system use only, and are not available to VxWorks users. [Table A-16](#) describes the routines that affect the contents of the events register.

Table A-16 **Event Register Routines**

Routine	Effects
eventReceive()	Clears or leaves the contents of the event register intact, depending on the options selected.
eventClear()	Clears the contents of the event register.
eventSend()	Copies events into the event register.

Table A-16 **Event Register Routines** (cont'd)

Routine	Effects
semGive()	Copies events into the event register, if a task is registered with the semaphore.
msgQSend()	Copies events into the event register, if a task is registered with the message queue.

A.4.7 VxWorks Events API

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

A.4.8 Show Routines

For the purpose of debugging systems that make use of events, the **taskShow**, **semShow**, and **msgQShow** libraries display event information.

The **taskShow** library displays the following information:

- the contents of the event register
- the desired events
- the options specified when **eventReceive()** was called

The **semShow()** and **msgQShow()** libraries display the following information:

- the task registered to receive events
- the events the resource is meant to send to that task
- the options passed to **semEvStart()** or **msgQEvStart()**

A.5 Watchdog Timers

VxWorks includes a watchdog-timer mechanism that lets any C routine be connected to a specified time delay. Watchdog timers are maintained as part of the system clock ISR. For information about POSIX timers, see [5.2 POSIX Clocks and Timers](#), p.90.

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

Restrictions on ISRs apply to routines connected to watchdog timers. The routines in [Table A-17](#) are provided by the **wdLib** library.

Table A-17 **Watchdog Timer Calls**

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

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

Example A-4 **Watchdog Timers**

```
/* Creates a watchdog timer and sets it to go off in 3 seconds.*/  
  
/* includes */  
#include "vxWorks.h"  
#include "logLib.h"  
#include "wdLib.h"  
  
/* defines */  
#define SECONDS (3)
```

```
WDOG_ID myWatchDogId;
task (void)
{
    /* Create watchdog */
    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);

    /* Set timer to go off in SECONDS - printing a message to stdout */
    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                "Watchdog timer just expired\n") == ERROR)
        return (ERROR);
    /* ... */
}
```

A.6 Interrupt Service Routines

Hardware interrupt handling is of key significance in real-time systems, because it is usually through interrupts that the system is informed of external events. For the fastest possible response to interrupts, VxWorks runs interrupt service routines (ISRs) in a special context outside of any task's context. Thus, interrupt handling involves no task context switch. [Table A-18](#) lists the interrupt routines provided in **intLib** and **intArchLib**.

Table A-18 **Interrupt Routines**

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

A.6.1 Interrupt Stack

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

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

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

A.6.2 Writing and Debugging ISRs

There are some restrictions on the routines you can call from an ISR. For example, you cannot use routines like **printf()**, **malloc()**, and **semTake()** in your ISR. You can, however, use **semGive()**, **logMsg()**, **msgQSend()**, and **bcopy()**.

A.6.3 Special Limitations of ISRs

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

Table A-19 Routines that Can Be Called by Interrupt Service Routines

Library	Routines
bLib	All routines
errnoLib	errnoGet() errnoSet()
fppArchLib	fppRestore() fppSave()

Table A-19 **Routines that Can Be Called by Interrupt Service Routines** (cont'd)

Library	Routines
intLib	intContext() intCount() intVecGet() intVecSet()
intArchLib	intLock() intUnlock()
logLib	logMsg()
lstLib	All routines except lstFree()
mathALib	All routines, if fppRestore() or fppSave() is used
msgQLib	msgQSend()
pipeDrv	write()
rngLib	All routines except rngCreate() and rngDelete()
selectLib	selWakeup() selWakeupAll()
semLib	semGive() except mutual-exclusion semaphores semFlush()
sigLib	kill()
taskLib	taskIdDefault() taskIdVerify() taskIsReady() taskIsSuspended() taskPriorityGet() taskPrioritySet() taskResume() taskSuspend() taskTcb()
tickLib	tickAnnounce() tickGet() tickSet()

Table A-19 **Routines that Can Be Called by Interrupt Service Routines** (cont'd)

Library	Routines
tyLib	tyIRd() tyITx()
vxLib	vxMemProbe() vxTas()
wdLib	wdCancel() wdStart()

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

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

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

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

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

All VxWorks utility libraries, such as the linked-list and ring-buffer libraries, can be used by ISRs. As discussed earlier ([A.2.6 Task Error Status: *errno*](#), p.285), the global variable **errno** is saved and restored as a part of the interrupt enter and exit code. Thus, **errno** can be referenced and modified by ISRs as in any other code. [Table A-19](#) lists routines that can be called from ISRs.

A.6.4 Exceptions at Interrupt Level

When a task causes a hardware exception such as an illegal instruction or bus error, the task is suspended and the rest of the system continues uninterrupted. However, when an ISR causes such an exception, there is no safe recourse for the system to handle the exception. The ISR has no context that can be suspended. Instead, VxWorks stores the description of the exception in a special location in low memory and runs a system restart.

The VxWorks boot programs test for the presence of the exception description in low memory and if it is detected, display it on the system console. The **e** command in the boot ROMs re-displays the exception description.

One example of such an exception is the following message:

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

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

A.6.5 Reserving High Interrupt Levels

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



CAUTION: Some hardware prevents masking certain interrupt levels. Check the hardware manufacturer's documentation.

A.6.6 Additional Restrictions for ISRs at High Interrupt Levels

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

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



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

A.6.7 Interrupt-to-Task Communication

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

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

B

PowerPC Considerations

- [B.1 Introduction 329](#)
- [B.2 Building Applications 330](#)
- [B.3 Memory Management Unit 332](#)
- [B.4 Protection Domains \(PowerPC 60x\) 337](#)
- [B.5 Architecture Considerations 337](#)

B.1 Introduction

This documentation includes information specific to VxWorks 653 for PowerPC targets.

For information on BSP-specific issues and device drivers, see the relevant BSP documentation.

For information on configuring and building VxWorks 653 systems, see the *VxWorks 653 Configuration and Build User's Guide*.

B.2 Building Applications



NOTE: The GNU compiler for PowerPC conforms to the Embedded Application Binary Interface (EABI). Therefore, type-checking is more rigorous than for some other architectures.

If you customize your BSP or need to change how it is built, you may need the information in this section.

Defining the CPU-Type Configuration Variable (CPU)

Setting the CPU-type configuration variable (**CPU**) ensures that VxWorks 653 and applications are compiled with features enabled that are specific to the PowerPC. [Table B-1](#) shows the values for the **CPU** variable for supported processors.

Table B-1 **CPU-Type Configuration Variable: Values for Supported Processors**

CPU Value	Processor
PPC603	PowerPC 82xx
	PowerPC 8349E
PPC604	PowerPC 750
	PowerPC 74xx
	Only the mpc74xx microprocessor core is supported. The AltiVec technology implemented in the PowerPC 74xx Vector Unit is not supported.
	PowerPC 8641D
	Only single core is supported.
PPC85XX	PowerPC 8560

As an example, to specify **CPU** for a PowerPC 750 in a header or source file, include the following line in the file:

```
#define CPU PPC604
```

Setting Compiler Options

The following is an example of a command line that compiles an application (*applic.language_id*) that is to run on a PowerPC 750:

```
% ccppc -O2 -mcpu=604 -IinstallDir/target/h -fno-builtin \  
-fno-for-scope -DCPU=PPC604 -c applic.language_id -g
```

The options have the following meaning:

-O2

Optional. Performs level-2 optimization.

-mcpu=604

Required. Produces code for the specified PowerPC architecture.

Other values are **603** and **8540**.

-IinstallDir/target/h

Required. Gives access to VxWorks 653 include files. Include additional **-I** options to specify additional header files.

-fno-builtin

Required. Uses library calls even for common library routines.

-fno-for-scope

Required. Lets the scope of variables declared in a **for** loop be outside the **for** loop.

-DCPU=PPC604

Required. Instructs VxWorks 653 to include code for the specified architecture. For other values, see [Table B-1](#).

-c

Required. Causes the application to be compiled, but not linked.

applic.language_id

Required. Specifies the file or files to compile. For C files, specify a **.c** *language_id*. For C++ files, specify **.cpp**. The output is an unlinked object module in ELF format with the **.o** extension.

-g

Optional. Generates debug information for the application.

B.3 Memory Management Unit

The following sections supplement MMU information in [7.8.3 Managing Page-oriented Memory](#), p.153.

B.3.1 Enabling or Disabling Instruction MMUs and Data MMUs

The PowerPC distinguishes between an instruction MMU and a data MMU. You can enable or disable each separately.

B.3.2 Mapping Memory (PowerPC 60x)

The MMU for PowerPC 603 and 604 supports two models for mapping memory: the BAT model and the segment model. VxWorks 653 supports both.

BAT Model for Mapping Memory

The BAT model for memory mapping maps into a BAT register a memory block ranging in size from 128 KB to 256 MB.

A BAT register is two 32-bit words. The PowerPC 603 and 604 have eight BAT registers: four for the instruction MMU and four for the data MMU. The **sysBatDesc[]** data structure (defined in **sysLib.c**) handles configuring BAT registers. The initialization routines in the MMU library set the registers. By default, they are set to zero.

Segment Model for Mapping Memory

The segment model for memory mapping specifies the configuration for each memory page. For the PowerPC, memory pages are 4 KB.

The configuration and build facility generates a data structure (**sysMemCfgTbl[]**) from the region information in the XML configuration file and puts it in **configRecord.reloc**. The data structure describes the entire physical memory and consists of configuration constants for each page or group of pages.

B.3.3 Setting MMU Access Rights

MMU access rights for a shared data region depend on the following XML attributes:

- `/Module/SharedDataRegions/SharedData/SharedDataDescription/@SystemAccess`
- `/Module/Partitions/Partition/PartitionDescription/SharedDataRegion/@UserAccess`

MMU access rights for a shared I/O region depend on the following XML attributes:

- `/Module/SharedDataRegions/SharedData/SharedIODescription/@SystemAccess`
- `/Module/Partitions/Partition/PartitionDescription/SharedDataRegion/@UserAccess` (this is the same Xpath as for the shared data region)

For details, see the *VxWorks 653 Configuration and Build Reference*.

[Table B-2](#) shows the following PowerPC MMU access information for shared data regions and shared I/O regions:

- Allowable combinations of values for their associated **SystemAccess** and **UserAccess** attributes.
- Resulting values of the page protection (PP) field in the page table entry (PTE).
- Actual meanings of the PP field. (Because VxWorks 653 programs the no-execute (N) field of the segment register (SR) to allow execute access, the actual meaning of the PP field is different.)

Table B-2 **PowerPC MMU Access Information for Shared Data Regions and Shared I/O Regions**

SystemAccess	UserAccess	PP Value	PP Meaning
READ_WRITE	NONE	0	Supervisor read-write-execute User none
READ_WRITE	READ_ONLY	1	Supervisor read-write-execute User read-execute

Table B-2 **PowerPC MMU Access Information for Shared Data Regions and Shared I/O Regions** (cont'd)

SystemAccess	UserAccess	PP Value	PP Meaning
READ_WRITE	READ_WRITE	2	Supervisor read-write-execute User read-write-execute
READ_ONLY	READ_ONLY	3	Supervisor read-execute User read-execute

For details on memory page attributes, see the [7.8.3 Managing Page-oriented Memory](#), p.153.

B.3.4 Setting MMU Cache Attributes

[Table B-3](#) lists valid combinations of MMU cache attributes for the PowerPC. You can specify any of the attributes based on 4-KB pages. For more information on cache attributes, see the information on programming environments in the *PowerPC Microprocessor Family*.

MMUs in the Guarded State

If the MMU is in the guarded state, instructions and data cannot be accessed out of order. If the MMU is not in the guarded state, instructions and data can be accessed out of order. With the PowerPC, when the MMU is enabled, the guarded state is readable, but not executable.

MMUs in the Coherent State

If the MMU is in the coherent state, store operations by all processors to the same memory location are ordered, and no processor is able to observe any subset of those store operations as occurring in a conflicting order. If the MMU is not in the coherent state, the order in which store operations from different processors occur is undefined.

Table B-3 **PowerPC MMU Cache Attributes**

OFF	COPY-BACK	WRITE-THRU	GUARDED	COHERENCY
x				
x			x	
x				x

Table B-3 PowerPC MMU Cache Attributes (cont'd)

OFF	COPY-BACK	WRITE-THRU	GUARDED	COHERENCY
X			X	X
	X			
	X		X	
	X			X
	X		X	X
		X		
		X	X	
		X		X
		X	X	X

B

Determining the Size of Hash Tables (PowerPC 604)

PowerPC processors use a two-level translation table to store MMU translation information. Processors based on the PowerPC 604 use a hash table as a cache, where entries are copies of entries in the translation table. These processors handle TLB reload operations faster if the MMU translation information is fetched from the hash table rather than from the translation table. As a consequence, for maximum performance, the size of the hash table must be large enough to hold all the translation entries in the translation table. The size of the hash table depends on the total memory to be mapped. The larger the memory to be mapped, the larger the hash table needs to be. The VxWorks 653 implementation of the segment model follows the recommendations given in the *PowerPC Microprocessor Family*. The total size of the memory to be mapped is calculated when the MMU library is initialized, allowing the size of the hash table to be dynamically determined. [Table B-4](#) shows the correspondence between the total memory to map and the size of the hash table.

Table B-4 PowerPC 604 MMU Hash Table Size

Total Memory to Map	MMU Hash Table Size
8 MB or less	64 KB
16 MB	128 KB

Table B-4 **PowerPC 604 MMU Hash Table Size** (cont'd)

Total Memory to Map	MMU Hash Table Size
32 MB	256 KB
64 MB	512 KB
128 MB	1 MB
256 MB	2 MB
512 MB	4 MB
1 GB	8 MB
2 GB	16 MB
4 GB	32 MB

Resizing and Moving Hash Tables (PowerPC 604)

To change the size of the MMU hash table, set the value of the **USER_HASH_TABLE_SIZE** configuration parameter with the following command:

```
prj domParameterValueSet -p coreOsDirectory USER_HASH_TABLE_SIZE newSize
```

where *newSize* is 2^n , $16 \leq n \leq 25$.

You can change the size of the hash table without moving it.

To change the location, change the value of the **USER_HASH_TABLE_ADDRESS** configuration parameter with the following command:

```
prj domParameterValueSet -p coreOsDirectory USER_HASH_TABLE_ADDRESS newAddress
```

By default, the hash table is allocated from the kernel heap. The hash table must align on an address (*newAddress*) that is an integral multiple of the hash table's size.

If you move the hash table, you must reserve memory for the table by creating a **kernelRegion** element in the XML configuration file. For details, see the *VxWorks 653 Configuration and Build Reference*.

B.3.5 ELF-Specific Tools

The GNU compiler provides the PowerPC-specific **objcopyppc** command. For details, see the reference entry for **objcopy** in the *GNU Binary Utilities*.

B.3.6 Detecting NULL Pointer Dereferences

The implementation for detecting NULL pointer dereferences in the PowerPC means that accesses to the first 16 KB of memory generate an exception. Accesses include read, write, or execute operations in supervisor or user mode.

B.4 Protection Domains (PowerPC 60x)

The implementation of virtual-memory support for the PowerPC 603 and 604 means that VxWorks 653 modules for the PowerPC 60x family cannot consist of more than a specific number of protection domains.

To reach the maximum number of protection domains that can be present in a VxWorks 653 module, change the configuration parameter (PD_MAX_NUMBER_OF_PDS) from its default value of 64 to the maximum as shown in [Table B-5](#).

Table B-5 **Maximum Number of Protection Domains**

Processor Family	Maximum Number of Protection Domains
PowerPC 603	4096
PowerPC 604	2016
PowerPC 85xx	Limited only by available memory

B.5 Architecture Considerations

This section describes characteristics of PowerPC processors that affect VxWorks 653 modules.

For comprehensive documentation of PowerPC architectures, see the appropriate Motorola microprocessor user's manual or the IBM user's manual.

Processor Mode

The PowerPC supports supervisor mode and user mode.

24-bit Addressing

To conform to the Embedded Application Binary Interface (EABI) standard, the PowerPC limits its relative addressing to 24-bit offsets.

Byte Order

VxWorks 653 for the PowerPC uses big-endian byte order.

PowerPC Registers

The Application Binary Interface (ABI) and the EABI protocols define PowerPC conventions on register usage, stack-frame formats, parameter passing between routines, and other factors involving code interoperability. VxWorks 653 for the PowerPC follows these protocols.

[Table B-6](#) shows PowerPC register usage in VxWorks 653.

Table B-6 **PowerPC Registers**

Register	Description	Volatile or Non-Volatile
fpr0	Floating-point register.	Volatile
fpr1	Floating-point register for passing parameters and returning values.	Volatile
fpr2 - fpr8	Floating-point registers for passing parameters.	Volatile
fpr9 - fpr13	Floating-point registers.	Volatile
fpr14 - fpr31	Floating-point registers for local variables.	Non-volatile
gpr0	Register for routine linkage.	Volatile
gpr1	Stack-frame pointer. It is always valid.	Non-volatile

Table B-6 **PowerPC Registers** (cont'd)

Register	Description	Volatile or Non-Volatile
gpr2	Register for the pointer to the second small data area (_SDA2_BASE_)	Volatile
gpr3, gpr4	Registers for passing parameters and returning values.	Volatile
gpr5 - gpr10	Registers for passing parameters.	Volatile
gpr11 - gpr12	Registers for routine linkage.	Volatile
gpr13	Register for the pointer to the small data area (_SDA_BASE_).	Non-volatile
gpr14 - gpr30	Registers for local variables.	Non-volatile
gpr31	Register for local variables or environment pointers.	Non-volatile

HI and HIADJ Macros

The **HI** and **HIADJ** macros are used in PowerPC assembly code. **HI(x)** is the high-order 16 bits of the value **x**. **HIADJ(x)** is the high-order 16 bits adjusted by bit 15. If bit 15 is set, the value is adjusted by adding 1.

You must use **HIADJ(x)** if the low-order 16 bits are to be used with an instruction that interprets them as a signed quantity (for instance **addi** and **lwz**). If the low-order bits are used in an instruction that interprets them as an unsigned quantity (for instance **ori**), use **HI**.

For example, **addi** uses a signed quantity, so **HIADJ** is the correct macro:

```
lis    rx, HIADJ(VALUE)
addi   rx, rx, LO(VALUE)
```

And, **ori** uses an unsigned quantity, so **HI** is the correct macro:

```
lis    rx, HI(VALUE)
ori    rx, rx, LO(VALUE)
```

Cache and Kernel Heap

To ensure cache coherency, the kernel heap must be created in cacheable mode: copy-back or write-through.

With protection-domain support, the location and size of the kernel heap is defined by the `KERNEL_MEM_POOL_START` and `KERNEL_MEM_POOL_SIZE` parameters. The initial cache mode is defined in the XML configuration file.

If protection-domain support is not included, the kernel heap extends up to user reserved memory, which is the value returned by `sysMemTop()`.

You can allocate non-cacheable buffers from the kernel heap. Routines that can mark buffers non-cacheable (such as `cacheDmaMalloc()` and `cacheDmaFree()`) work properly. It is recommended that you use these sorts of routines. However, if you need to manually alter cache mode for a buffer through the virtual-memory interface, you must restore the previous state before you release the buffer.

If you need to disable caching for all memory—for example, when debugging—undefine the `USER_D_CACHE_ENABLE` and `USER_I_CACHE_ENABLE` macros, rather than use the MMU to mark each individual page non-cacheable.

Floating-Point Routines

Table B-7 lists the floating-point routines that are available for PowerPC processors. A subset is optimized using Motorola libraries.

Table B-7 Floating-Point Routines

Routine	Optimized Using Motorola Libraries
<code>acos()</code>	Yes
<code>asin()</code>	Yes
<code>atan()</code>	Yes
<code>atan2()</code>	Yes
<code>ciel()</code>	
<code>cos()</code>	Yes
<code>cosh()</code>	

Table B-7 **Floating-Point Routines** (cont'd)

Routine	Optimized Using Motorola Libraries
<code>exp()</code>	Yes
<code>fabs()</code>	
<code>floor()</code>	
<code>fmod()</code>	
<code>log()</code>	Yes
<code>log10()</code>	Yes
<code>pow()</code>	Yes
<code>sin()</code>	Yes
<code>sinh()</code>	
<code>sqrt()</code>	Yes
<code>tan()</code>	
<code>tanh()</code>	

Routines Not Available

The following floating-point routines are not available on PowerPC processors:

- `cbrt()`
- `infinity()`
- `rint()`
- `iround()`
- `log2()`
- `round()`
- `sincos()`
- `trunc()`

No single-precision routines are available for PowerPC processors.

Support for Floating-Point Exceptions in Partitions

Floating-point exceptions are supported for PowerPC processors and support is enabled by default. Only partitions support floating-point exceptions. Core OS tasks do not.

To disable support for floating-point exceptions for a particular partition, set the partition's **fpExcEnable** parameter to **false** in the XML configuration file at configuration and build time. For details, see the *VxWorks 653 Configuration and Build Reference*.

Machine State Register

When support for floating-point exceptions is enabled, the machine state register (MSR) is saved and restored on each process context switch.

Only a supervisor-level task can change the MSR. An application cannot alter it.

When support for floating-point exceptions is enabled, the MSR for the partition is set to floating-point precise mode; that is, MSR[FE0] and MSR[FE1] are set to 1.

If support for floating-point exceptions is disabled, the MSR for the partition is set to floating-point exceptions-disabled mode; that is, MSR[FE0] and MSR[FE1] are set to 0. In this mode, floating-point exceptions return a predefined value instead of causing an exception.

Floating-Point Status and Control Register

An application can modify settings for the floating-point status and control register (FPSCR) for its partition. For example, an application might enable or disable types of floating-point exceptions, such as underflow and overflow exceptions.

If changes are made to the register for an application task, the changes affect only that thread of execution. Changes are saved for that task's context when a context switch occurs. Context switches include a task context switch within a partition or a partition context switch.

By default, the value of the FPSCR is 0x000000D0. This enables the following bits and their associated exceptions:

- VE (invalid)
- ZE (divide by zero)
- OE (overflow)

Shared Library Support (PowerPC 604)

The MMU implementation for the PowerPC 604 forces limitations in shared-library support in VxWorks 653. The shared-library size requirements are the same as in other architectures, and theoretically the maximum number of shared libraries is the same. But for the PowerPC 604, the following applies:

- The time to attach and detach shared libraries is longer.
- The use of hash-table entries is increased, not only because of more shared libraries, but because of attachments made to them.

If your VxWorks 653 module has many attachments to shared libraries, you may need to increase the default size of the PowerPC 604 MMU hash table. Increasing the size ensures that translation information is fetched from the hash table instead of from the two-level translation tables, thereby guaranteeing the best performance. You can monitor hash-table usage by calling **mmuPpcHashTblShow()** from the shell. To use this routine, you must include the **INCLUDE_KERNEL_SHOW** component in the kernel domain. For more information, see [Resizing and Moving Hash Tables \(PowerPC 604\)](#), p.336 and the reference entry for **mmuPpcHashTblShow()**.

B

Debugging

Caches

If you need to disable caching for all memory—for example, when debugging—undefine the **USER_D_CACHE_ENABLE** and **USER_I_CACHE_ENABLE** macros, rather than use the MMU to mark each individual page as non-cacheable.

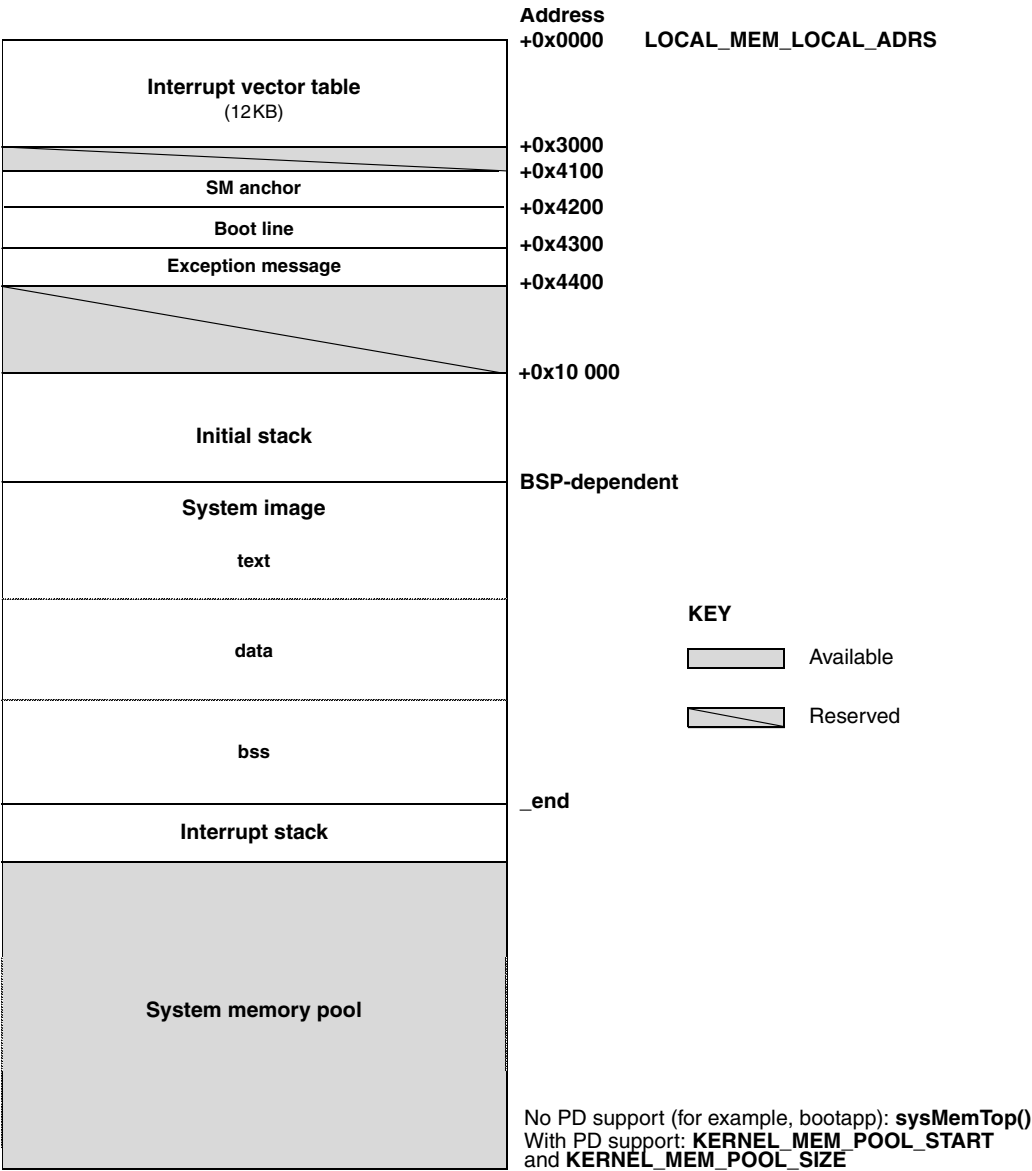
Memory Layout

The VxWorks 653 virtual memory layout is the same for all PowerPC processors. [Figure B-1](#) shows the memory layout, labeled as follows:

Interrupt vector table	Table of exception and interrupt vectors.
SM anchor	Anchor for the shared memory network objects (if there is shared memory on the board).
Boot line	ASCII string of boot parameters.
Exception message	ASCII string specifying the fatal-exception message.
Initial stack	Initial stack for usrInit() until usrRoot() is allocated a stack.
System image	VxWorks 653 itself (three sections: text, data, bss). The entry point for VxWorks 653 is at the start of this region, whose address depends on the BSP. The entry point for most supported BSPs is 0x100,000. For a list of supported BSPs, see the Platform release notes.
Interrupt stack	The size (in bytes) of the interrupt stack is defined by the ISR_STACK_SIZE parameter. For information on setting parameters, see the <i>VxWorks 653 Configuration and Build Guide</i> . For information on a particular parameter, see the <i>VxWorks 653 Configuration and Build Reference</i> .
System memory pool	Size depends on the size of the module image. The sysMemTop() routine returns the address of the end of the free memory pool.

Addresses shown in [Figure B-1](#) are relative to the start of memory for a particular board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined by the **LOCAL_MEM_LOCAL_ADRS** parameter. For information on setting parameters, see the *VxWorks 653 Configuration and Build Guide*. For information on a particular parameter, see the *VxWorks 653 Configuration and Build Reference*.

Figure B-1 VxWorks 653 System Memory Layout for the Core OS (PowerPC)



B

C

Glossary

acceptance

Acceptance is the acknowledgement by a certification authority that the ARINC 653 module, application, or system meets its defined requirements.

ACE

ACE: Agent for the Certified Environment.

AFDX

AFDX: Avionics Full Duplex Switched Ethernet. It is defined by the ARINC 664 specification, Part 7.

alarm

In the context of health monitoring, an alarm is an event. See also message.

AMIO

Application multiplexed I/O (AMIO) allows you to provide input to and view output from multiple partitions over a single serial connection.

APEX

APEX: Application/Executive. The general-purpose interface between an OS and application software, specified by the ARINC 653 specification. The specification includes the list of services that lets the application control scheduling, communication, and status information of its internal processing elements.

APEX port

APEX port: see port.

API

API: application programming interface.

application

An application is a collection of software components that together perform a specific function in an embedded system. See also application partition.

application developer

An application developer develops one or more applications that will reside in a partition. This person or group may also be responsible for developing data binaries, which contain any databases used by the application. See also platform provider and system integrator.

application partition

An application partition is a partition that includes an application.

APPS

APPS: ARINC PPS. It is the module-wide scheduling scheme for partitions. This is a combination of ARINC 653 scheduling (TPS) and PPS scheduling in which the PPS scheme is used during idle time within the TPS scheme. The scheduling scheme applies to all PPS-enabled partitions in the module.

ARINC 653

ARINC 653 refers to ARINC Specification 653: "Avionics Application Software Standard Interface."

ARINC 653 scheduling

ARINC 653 scheduling is the scheduling that is specified by the ARINC 653 specification. It is time-preemptive scheduling (TPS). See also APPS scheduling and PPS scheduling.

ARINC PPS

ARINC PPS: see APPS scheduling.

black box

A black box is a set of configuration parameters that represent the memory requirements of an application, a shared library, or the core OS. The use of black boxes allows a VxWorks 653 module to be configured before all the applications and libraries are available. Applications, libraries, and the core OS must fit within the memory limits set by their black boxes.

board support package

BSP: board support package. It provides the libraries required to support a platform on a particular board. The BSP, along with the kernel and user-supplied extensions, make up the core OS.

BSP

BSP: see board support package.

BSP developer

A BSP developer is a person or organization responsible for the development of a board support package.

BSS

BSS: block started by symbol. It is a data section in an ELF file that contains uninitialized global and static variables that are zeroed.

build spec

A build spec specifies compiler and linker options to produce particular output, such as cert, debug, or release.

callback routine

In the context of health monitoring, a callback routine is called when an event arrives at a partition health monitor task or module health monitor task. It is called before the handler for the given event is called.

CDF

CDF: component description file. It has the **.cdf** extension. It uses the component description language (CDL) to name and give values to the parameters of VxWorks 653 components.

cert

cert is the build spec that produces a certifiable image.

certifiable

An image that is certifiable can be certified to a specific level of the DO-178B avionics software standard.

certifiable subset

A certifiable subset is a subset of the core OS or a partition OS that can be certifiable to Level A of the DO-178B avionics software standard.

certification

Certification refers to certification to a specific level of the DO-178B avionics software standard.

channel

A channel defines a logical link between one source port and one or more destination ports. It also defines the message transfer mode and the characteristics of the messages. Channels are used for interpartition communication, which can be between local partitions, pseudo-partitions, or both. Channels conform to the ARINC 653 specification.

COIL

COIL: core OS interface library. A partition OS that provides a library of routines independent of the vThreads partition OS. The library supports the management of interrupts and exceptions, device I/O, interpartition messaging, and injection of health monitoring events.

COIL partition

A COIL partition is a partition whose partition OS is based on COIL. See also vThreads partition.

cold restart

A cold restart occurs when a module or partition is restarted and all data is reloaded. A cold restart takes longer than a warm restart.

configlette

A configlette is a component or part of a component that is distributed in source form, allowing compile-time parameters to be set when the component is included in a build.

configuration parameter

A configuration parameter is used to change the configuration of a VxWorks 653 component.

configuration record

A configuration record is a record of the information that makes up the configuration of a VxWorks 653 module or a part of it. Configuration records include both the system configuration record and user configuration records.

core OS

The core OS is the core operating system for a VxWorks 653 module. It provides fundamental operating system services and schedules partitions.

core OS interface library

Core OS interface library: see COIL.

CPU page size

The CPU page size is the smallest addressable unit of memory for the MMU. It is also called MMU page size. The page size depends on the CPU and is generally not configurable.

cross-development tools

Cross-development tools are programs that run on a host computer (running, for example, Windows or UNIX) and that are used to develop, debug, or control software running on an embedded processor that is running a real-time operating system (for example, VxWorks 653). For VxWorks 653, the cross-development tools are based on Workbench. See also run-time software.

current partition

The current partition is the partition that is running. In an APPS scheduling environment, the current partition and the TPS partition may not be the same.

default schedule

The schedule that will be run when the module is started.

destination port

A destination port is one of possibly many ports at the receiving end of a channel. See also source port.

direct-access port

A direct-access port is a type of pseudo-port that does not use software buffering. Buffering support is assumed to be provided by the communications hardware.

DO-178B

DO-178B: “Software Considerations in Airborne Systems and Equipment Certification.” The avionics software standard developed by RTCA.

domain

A domain is a software container. Each element of a VxWorks 653 module—the core OS (kernel), partitions (applications), shared libraries, and shared data regions—exists in a domain.

dynamic memory allocation

Dynamic memory allocation refers to allocating memory from the heap at run-time.

EABI

EABI: Embedded Application Binary Interface.

ELF

ELF: Executable and Linking Format. It is an object module format used to encapsulate compiled software.

error handler process

See process health monitor.

event

In the context of health monitoring, an event is the base unit that is injected into the event handling framework. It represents an alarm or a message, depending on the event code.

event code number

In the context of health monitoring, an event code number is the value of the event code, as defined in the **HM_CODE** enumeration type in **hmTypes.h**.

event queue

The module health monitor table and partition health monitor table each have an event queue. The module and partition health monitor event queues are sometimes called, simply, the module and partition health monitor queues. An event queue holds the events that have been dispatched to its associated health monitor for handling. Event queues are serviced before health monitor notification queues are serviced.

FAA

FAA: U.S. Federal Aviation Administration.

FIFO

FIFO: first-in, first-out queuing.

FPSCR

Floating-point status and control register.

global file descriptor

Global file descriptors (standard in, standard out, and standard error) are available to all tasks in a partition. Their global assignment is controlled by the **ioGlobalStdSet()** and **ioGlobalStdGet()** routines, but may be overridden by the **ioTaskStdSet()** and **ioTaskStdGet()** routines.

GUI

GUI: graphical user interface.

health monitor

Health monitoring provides a framework to raise and handle events (which can be alarms or messages) in a VxWorks 653 module. Alarms are injected to represent faults, and handlers provide the opportunity to perform recovery actions. See module health monitor, partition health monitor, process health monitor, and system health monitor.

hosted function supplier

Hosted function supplier: see application developer.

IDE

IDE: integrated development environment.

injection

Injection is the act of creating a health monitor alarm event or message event.

interface subset

An interface subset defines part of the interface of a shared library. The use of interface subsets allows you to reuse parts of the interface definition among libraries that share some parts of their interface. For example, two different **vThreads** libraries containing different components would share the core **vThreads** interface.

interrupt level

Saying an event is injected at an interrupt level means the event is injected from an interrupt execution context.

ISR

ISR: interrupt service routine.

jitter

Jitter is a variation or deviation in the frequency of an expected occurrence. See also partition switch jitter.

kernel

Kernel is another term for the core OS.

kernel I/O region

A kernel I/O region is a region of target memory that corresponds to the address of an I/O device on the target and can be accessed only by the core OS.

Level A

Level A is the highest certification level for the DO-178B software standard.

loadable shared data region

A loadable shared data region is a data source, such as a database, that can be loaded into a shared data region as part of the module payload.

local partition

A local partition is a partition that is local to a VxWorks 653 module. Unless it might be confused with a pseudo-partition, it is called, simply, a partition.

local port

A local port is a port that is attached to a local partition. Unless it might be confused with a pseudo-port, it is called, simply, a port. See also null port.

log queue

The module health monitor and partition health monitor each have a log queue (sometimes called simply a log). Health monitor messages are always logged, whereas alarms are logged only if health monitor logging is enabled. If an event is injected from within a partition (**HM_PROCESS_MODE** or **HM_PARTITION_MODE**), the event is logged to the partition health monitor log. If the event is injected from outside the partition (**HM_MODULE_MODE**), the event is logged to the module health monitor log.

major frame

Each schedule consists of a major frame, which is divided into a series of variable-length minor frames.

message

In the context of health monitoring, a message is an event. See also alarm.

minor frame

Each schedule consists of a major frame, which is divided into a series of variable-length minor frames. Each minor frame defines the partition to run, its allowed duration, and whether or not the minor frame is a release point.

MMU

MMU: memory management unit.

module

A module is the “system” controlled by one RTOS, and in VxWorks 653, that RTOS is the core OS.

module health monitor

The module health monitor is present in parallel with all partitions in a VxWorks 653 module, and hence all partition health monitors in the module. The module health monitor is not part of any partition window and has priority over all partitions. The module health monitor resides in the core OS. It is associated with the module health monitor table, which among other things, defines notification queues, a log queue, and an event queue. See also system health monitor, partition health monitor, and process health monitor.

MSR

Machine state register.

namespace

An XML namespace provides a unique identifier which can be associated with an XML element by means of a prefix. The namespace uniquely identifies the XML schema in which the element is defined.

NMI

NMI: non-maskable interrupt.

normal mode

Normal mode is the partition mode during which processes or threads are scheduled. (Other partition modes include idle, cold start, and warm start.)

notification

In the health monitoring context, notification is the act of informing another partition health monitor or the module health monitor of an event that has occurred in a given partition.

notification queue

The module health monitor table and partition health monitor table each have notification queues, one for each partition that wants to accept notification of events. Notification queues are serviced after health monitor event queues are serviced.

null port

A null port is a port that is created at system initialization time, but is not used. It is always considered to be empty when read from and have space when written to. A null port can be attached to a partition, the core OS, or a pseudo-partition. See also local port and pseudo-port.

NVM

NVM: non-volatile memory.

online-loaded partition

With online-loaded partitions, the core OS does not install the partition code from flash or RAM into its final domain location in RAM as it does during the system initialization phase for regular partitions. Instead, an empty application domain is created for an online-loaded partition during the core OS initialization phase. The code of the online-loaded partition is made available to the core OS only at a later stage. In some cases this may not be until after all the regular partitions are already running.

OS

OS: operating system.

partition

A partition is a container for an application. An application running in a partition cannot interfere with applications in other partitions or with the core OS.

partition direct-access port

A partition direct-access port is a type of direct-access port residing in a partition. A partition direct-access port can communicate only with a local port in the application resident in the partition.

partition health monitor

The partition health monitor is the health monitor that is present in parallel with vThreads to handle vThreads partition errors and events that may affect the operation of vThreads within the partition. The partition health monitor is scheduled as part of the partition window. It is associated with the partition health monitor table, which among other things, defines notification queues, a log queue, and an event queue. See also system health monitor, module health monitor, and process health monitor.

partition OS

A partition OS is a user-level software library running within a partition that provides operating system services to the partition. See also vThreads and COIL.

partition OS scheduler

The partition OS scheduler is the scheduler in a partition OS that allocates CPU time to threads in the partition. It is a priority-preemptive scheduler and is not related to the ARINC schedule.

partition port

Partition port: see local port.

partition scheduler

The partition scheduler is the scheduler in the core OS that allocates CPU time to partitions, allowing CPU time to become available to threads in those partitions. By default, the partition scheduler uses ARINC 653 (TPS) scheduling, but can optionally schedule designated partitions with APPS scheduling. See also partition OS scheduler.

partition switch jitter

Partition switch jitter is a variation or deviation in the configured partition switching schedule. For example, partition switch jitter might be caused by hardware latencies or by the core OS locking interrupts.

partition window

A partition window is the time in which a partition is allowed to run before being scheduled out.

payload

A payload is an image file (or files) that contains the code for a VxWorks 653 module in a form that is suitable for running on a target.

payload region

A payload region is the region of RAM or ROM where a payload is loaded.

periodic process

A periodic process is a process within a partition that is run on a schedule based on the passage of wall clock time. That is, the countdown to the next invocation of a periodic process runs even when the partition itself is not scheduled.

PersistentBSS

A BSS section that is persistent across a warm restart.

platform

A platform is software on which applications can be built and from which a VxWorks 653 module can be developed.

platform provider

A platform provider is responsible for configuring the base system on which application developers will build their applications.

port

A port is one end of a channel, which is used for interpartition communication. Ports have attributes, for example, direction (source or destination), mode (queuing or sampling), protocol (receiver discard, sender block, or none), and refresh rate. Ports conform to the ARINC 653 specification and its APEX interface and are also called APEX ports. See also pseudo-port.

POS

POS: See partition operating system.

POSIX

POSIX: Portable Operating Systems Interface. In this documentation, POSIX refers to the standard for real-time extensions (1003.1b), which specifies a set of interfaces to OS facilities. The POSIX API can be included in a vThreads partition if the APEX API is not included.

PPS

PPS: priority-preemptive scheduling. It allows for scheduling of partitions in a module-wide priority-preemptive scheme during the idle time within an ARINC 653 (TPS) schedule. See also APPS scheduling.

PPS-enabled

A PPS-enabled partition is a partition that is configured to indicate that it should be considered during APPS scheduling.

preemption locking

Preemption locking disables the scheduling of processes, threads, or tasks, and only the current process, thread, or task can be run until it decrements the lock level back to zero.

priority-preemptive scheduling

Priority-preemptive scheduling: see PPS.

process

Process is the APEX term for a thread. In the vThreads context, the term thread is preferred. See also task.

process health monitor

The process health monitor is the health monitor that is present within vThreads to handle process-related errors and events. It is also known as the error handler process. See also system health monitor, module health monitor, and partition health monitor.

pseudo-partition

A pseudo-partition is a communications object that is outside a VxWorks 653 module. See also local partition and pseudo-port.

pseudo-port

The term pseudo-port applies generally to any port that represents a data source or destination outside the current module. The term pseudo-port is also used in a more restrictive sense for a type of pseudo-port that uses software buffering. In this sense it is contrasted with direct-access port which is a type of pseudo-port that does not use software buffering. See also local port and null port.

queuing port

A queuing port is a port in queuing mode. In queuing ports, messages are queued. A protocol manages the queues. See also sampling port.

RAM

RAM: random access memory.

RAM payload

A RAM payload is a payload that is designed to be downloaded into RAM on the target.

real-world time

Real-world time: see wall clock time.

receiver discard protocol

Receiver discard protocol is a port message protocol. If one of the channel's destination ports is full, the source port discards the message for that port. Therefore, if all the destination ports are full, the message might be lost. When a message is so discarded, the port's overflow flag is set to notify the application of the discarded (lost) message. See also sender block protocol.

refresh rate

The refresh rate (in seconds) indicates the maximum acceptable age of a valid message, from the time it was received by the port. It applies to destination sampling ports only.

release point

A release point is a way to synchronize a periodic process with the partition window of a partition. A periodic process spawned in a partition starts only at the next release point.

ROM

ROM: read-only memory.

ROM payload

A ROM payload is a payload that is designed to be installed in ROM on the target.

root element

The root element is the element of an XML document that contains all the other elements in the document.

RTCA

RTCA: Radio Technical Commission for Aeronautics. The private, not-for-profit corporation that develops recommendations on communications, navigation, surveillance, and air-traffic management issues. RTCA developed the DO-178B avionics software standard.

RTOS

RTOS: real-time operating system.

run-time software

Run-time software is the operating system and application software that together run on a target. See also cross-development tools.

sampling port

A sampling port is a port in sampling mode. In sampling ports, messages are not queued. A message remains in the source port until it is sent or overwritten. Each new message overwrites the previous one when it reaches the destination port and remains there until it is overwritten itself. Sampling ports have refresh rates. See also queuing port.

SAP port

A service access point (SAP) is a special kind of queuing port. It is different from a normal queuing port because it allows access to addressing information when sending and receiving messages. The SAP services are similar to the ARINC 653 queuing port services but has additional parameters to support address information. ARINC 653 Part 2, Supplement 2, defines two types of SAP services. Standard SAP services provide limited addressing capability to ensure that the source cannot alter its identity and that the destination is unambiguous. Extended

SAP services provide complete accessibility to addressing. VxWorks 653 supports standard SAP ports, but not extended SAP ports.

schedule

Schedules define how the core OS schedules partitions. Each schedule consists of a major frame.

scheduler

See partition scheduler and partition OS scheduler.

select operation

The select operation refers to calling **select()** to pend on a set of file descriptors.

sender block protocol

Sender block protocol is a port message protocol. A queuing message is sent to all the channel's destination ports. If any one is full, the message is queued in the source port in FIFO order. When the source port is full and if a timeout was specified, sender processes are blocked during the **SEND_QUEUING_MESSAGE** service. When a destination port is emptied, retransmission is attempted. Whether it succeeds depends on the state of the channel's other destination ports. See also receiver discard protocol.

service access point

Service access point: see SAP port.

shared data region

A shared data region (sometimes called a shared data domain) is a data region that can be used by applications within partitions to share data. Outside a shared data region, applications have no access to the data of other applications. See also loadable shared data region.

shared I/O region

A shared I/O region is a region of target memory that corresponds to the address of an I/O device on the target and can be shared by partitions and the core OS.

shared library

A shared library is a library that contains code that can be shared by multiple applications. See also system shared library.

shared library region

A shared library region is the area of RAM that holds a shared library.

source port

A source port is the one port at the sending end of a channel. See also destination port.

standard port

Standard port: see local port.

static module

A static module file is a fully located object file that has been compiled and linked for use in a VxWorks 653 module. A static module file has a .sm file extension.

straight-line code

Straight-line code is code that does not use threads.

system call

A system call is a call from a partition to the core OS.

system clock

System clock refers to the system clock for a VxWorks 653 module.

system configuration record

The system configuration record is the record of all the configuration parameters in a VxWorks 653 module. During the configuration process, configuration information is expressed in the **Module** configuration document. The build process produces a binary version of this information in **configRecord.reloc** or **configRecord.bin**.

system health monitor

The system health monitor is the dispatcher for the health monitoring system. See also module health monitor, partition health monitor, and process health monitor.

system heap

System heap refers to the heap for the core OS.

system initialization

System initialization refers to the initialization of a VxWorks 653 module.

system integrator

A system integrator is responsible for integrating the applications created by the application developers with the platform created by the platform provider to create the final module.

system memory

System memory refers to memory controlled by the core OS.

system object

A system object is an object created by the core OS (or vThreads) for use by the core OS (or vThreads). An example is a semaphore.

system resource

A system resource is a resource allocated by the core OS for use by the core OS.

system restart

System restart refers to restarting a VxWorks 653 module.

system shared library

A system shared library is a special shared library that contains the code for a partition OS.

system start

System start refers to starting a VxWorks 653 module.

target

The target is the board for which you are developing an embedded system.

task

A task is an execution context. In VxWorks 653, it refers to a core OS object. See also thread.

TCB

TCB: task control block. The structure that contains critical run-time information for a single task.

thread

A thread is an execution context. It is the preferred term for what is sometimes called a process. A thread is a programming unit contained within a vThreads partition. It runs concurrently with other threads of the same partition. See also task and process.

time-preemptive scheduling

Time-preemptive scheduling: see TPS.

TLB

TLB: translation look-aside buffer. It is a specialized cache that holds a table of physical addresses as generated from the virtual addresses that program code uses.

TPS

TPS: time-preemptive scheduling. It is also called ARINC 653 scheduling. See also APPS scheduling and PPS scheduling.

TPS partition

A TPS partition is the partition that has been scheduled to be run by the ARINC 653 (TPS) scheduler. In an APPS scheduling environment, the current partition and the TPS partition may not be the same.

trusted partition

From the point of view of a given partition, a trusted partition is a partition from which it will allow the health monitor to accept health monitor notifications on its behalf. Since health monitor notifications are processed in the time slice of the partition on whose behalf they are received, limiting the number of partitions that a partition trusts limits the effect of health monitor notifications on the partition's time allotment.

user configuration record

A user configuration record is a collection of data that can be used for configuring user extensions to the core OS.

user memory region

The user memory region is that area of RAM that is needed for memory other than health monitor logs, core OS configuration records, core OS memory, core OS page pools, core OS pools, ports, and RAM payload.

user partition OS

A user partition OS is a partition OS that is based on COIL, augmented to perform other functions that are required by the application.

VAL

VAL: vThreads abstraction layer. It is a layer of the core OS. When a vThreads partition makes a system call, it communicates with this layer. It is a concept internal to VxWorks 653.

validation

In XML terms, validation is a process that ensures that an XML file is well formed according to the rules of XML and adheres to the structure specified in the appropriate XML schema. Validation is performed by an XML validator.

VME

VME: Versa Module Europa. VME is an open-ended bus system that makes use of the Eurocard standard. The VME bus was intended to be a flexible environment, supporting a variety of computing-intensive tasks, and has become a popular protocol in the computer industry. It is defined by the IEEE 1014-1987 standard.

vThreads

vThreads is the priority-preemptive OS that serves as a partition OS.

vThreads partition

A vThreads partition is a partition whose partition OS is based on vThreads. See also COIL partition.

vThreads scheduler

vThreads scheduler: see partition OS scheduler.

VxWorks 5.5

VxWorks 5.5 is the Wind River operating system on which the vThreads partition OS of VxWorks 653 is based.

VxWorks 653

VxWorks 653 is the Wind River operating system that supports the ARINC 653 specification.

W3C

W3C refers to the World Wide Web consortium at www.w3.org.

wall clock time

Wall clock time is time as measured in the real world by the clock on the wall. (As opposed, for instance, to the time elapsed in a particular application's partition window.)

warm restart

A warm restart occurs when a module or partition is restarted but persistent data is retained, shortening the time required for the restart.

WDB

WDB refers to the Wind River debug agent.

Wind

Wind is the adjective applied to certain OS objects to distinguishes them from POSIX objects. For example, Wind semaphores distinguishes from POSIX semaphores.

WindSh

WindSh is a host shell.

Workbench

Workbench is the Wind River Workbench development environment.

worker task

A worker task is a core OS task that is associated with a specific partition. Worker tasks perform blocking operations (typically blocking I/O) on behalf of the partition they are associated with.

write-protect

To write-protect is to guard an entity by a mechanism that prevents it from being changed or erased. For example, memory can be write-protected by using an MMU.

XInclude

XInclude is a W3C standard for including one XML file in another.

XML

XML: Extensible Markup Language. It is a standard for defining markup languages.

XML attribute

An XML attribute is an additional piece of information added to an XML element in the form of a key-value pair.

XML declaration

The XML declaration identifies a file as an XML document and contains information such as the version of XML used and the character encoding used in the file.

XML document

A document written using XML syntax.

XML document type

An XML document type is the grammar of a particular XML file as defined by the applicable XML schema.

XML editor

An XML editor is a program that provides support for editing XML files. This usually includes support for inserting tags and for validating the file against an XML schema.

XML element

An XML document consists of XML elements, each of which may contain data content, other elements, or both. The elements allowed in a particular document type are determined by the applicable XML schema.

XML file

An XML file is an instantiation of an XML schema.

XML schema

An XML schema is a document that defines the structure of an XML document. It defines what elements are permitted in an XML document, the order and nesting of elements, and the types of data each element can contain.

XML schema file

An XML schema file is a file that contains all or part of the definition of an XML schema. An XML schema file can include other schema files by reference to construct a complete schema definition.

XPath

XPath is a W3C standard for expressing the location of an element or attribute in an XML file.

Index

Symbols

`_VTH_COM_INIT` macro 143

Numerics

24-bit addressing 338

A

access routines (POSIX) 92

actions

 health monitoring 210

addressing

 24-bit 338

AFDX 187

aioPxLib 90

alarm escalation

 health monitoring 210

alarm injection

 health monitoring 204

alarms

 health monitoring 196

allowable notifications 212

APEX

 blackboards

 queuing 80

 services 79

 state transition 81

 buffers 77

 channels 68

 deadline time 62

 events 83

 handling 84

 queuing 84

 state transitions 85

 inter-partition

 communication 67

APEX port RECV 22

APEX port SEND 22

application multiplexed I/O 256

APPS scheduling 172

 examples 178

 forcing idle 175

 idle time 175

 partition-scheduling routines 181

 pseudo-interrupts 177

 ticks and timeouts 176

 vThreads and 176

ARINC 664 187

askRegsGet() 280

asynchronous I/O (POSIX)

 aioPxLib 90

- attribute (POSIX)
 - prioceiling attribute 108
 - protocol attribute 108
- attributes (POSIX) 92
 - contentionscope attribute 93
 - detachstate attribute 93
 - inheritsched attribute 94
 - schedparam attribute 95
 - schedpolicy attribute 94
 - specifying 95
 - stackaddr attribute 92
 - stacksize attribute 92

B

- BAT
 - model (PPC 60x) 332
- binary semaphores 298, 301
- boot sequence, vThreads 25
- booting 8
- broadcast messages 68
- byte order 338

C

- C++ development
 - C and C++, referencing symbols between 125
 - exception handling 125
 - iostreams 127
 - Run-Time Type Information (RTTI) 126
 - Standard Template library (STL) 128
- C++ support
 - see also* iostreams (C++)
 - configuring 124
- cache
 - mode, selecting 150
- caches 340
 - heap 340
- callback
 - health monitoring 212
- cancelling threads (POSIX) 97

- certifiability
 - COIL 8
 - core OS 7
 - vThreads 7
- certification
 - vThreads 15
- channel mapping 74
- checkStack() 324
- client-server communications 312, 314
- CLOCK_REALTIME 90
- clockLib(1)
- clocks
 - see also* system clock
 - POSIX 90, 91
 - system 283
- code
 - interrupt service, *see* interrupt service routines
 - pure 289
 - shared 288
- code examples
 - message queues
 - attributes, examining (POSIX) 110
 - checking for waiting message (POSIX) 116, 120
 - POSIX 113
 - Wind 311
 - mutual exclusion 300
- semaphores
 - binary 300
 - named 106
 - recursive 304
 - unnamed (POSIX) 103
- tasks
 - deleting safely 282
 - round-robin time slice (POSIX) 101
 - scheduling (POSIX) 100
 - setting priorities (POSIX) 99
 - synchronization 300, 301
- threads
 - creating, with attributes 95
- watchdog timers
 - creating and setting 322

- COIL
 - certifiability 8
 - key features 4
 - overview 7
 - COIL I/O 264
 - blocking (no worker tasks) 267
 - blocking versus non-blocking (comparison with vThreads) 264
 - non-blocking (with worker tasks) 266
 - cold start/restart
 - partitions 160
 - system 157
 - COLD versus WARM restarts (vThreads) 27
 - configRecordLib 165
 - configuration
 - C++ support 124
 - signals 316
 - configuration tables
 - health monitoring 215
 - contexts
 - task 270
 - creating 277
 - conventions
 - task names 278
 - core OS
 - APPS scheduling 172
 - certifiability 7
 - key functions for COIL 6
 - key functions for vThreads 6
 - overview 6, 134
 - partition configuration record 135
 - partition support 165
 - partition-related components 165
 - partition-related routines 165
 - partitions 135
 - cold start/restart 160
 - warm restart 161
 - PPS scheduling 140, 172
 - scheduling partitions 169
 - shared data regions 143
 - shared libraries 142
 - system call permission bitmasks 138
 - system call stacks 141
 - system cold start/restart 157
 - system warm restart 159
 - TPS scheduling 170
 - core OS interface library *see* COIL
 - counting semaphores 102, 305
 - CPU type, defining 330
 - custom permission bitmask 140
- ## D
- daemons
 - telnet tTelnetd 293
 - data MMU 332
 - data structures, shared 294
 - debugging
 - error status values 285, 287
 - defining CPU type 330
 - delayed tasks 271
 - delayed-suspended tasks 271
 - deployed configurations
 - ROM payload image 157
 - development configurations
 - RAM payload image 158
 - device driver models
 - vThreads I/O 223
 - device I/O
 - vThreads 33
 - directed messages 68
 - dispatching
 - health monitoring 205
 - DO-178B certifiability
 - recommendations to ensure 10
 - VxWorks 653 9
 - DO-178B certification
 - vThreads 15
 - drivers
 - interrupt service routine limitations 326

E

- `__errno()` 286
- `errno` 285, 287, 326
 - and task contexts 286
 - example 287
 - return values 286
- error status values 285, 287
- event payload 200
- event reformatting
 - health monitoring 208
- `eventClear()` 320
- `eventReceive()` 320
- events
 - health monitoring 196
 - VxWorks events 321
 - API 321
 - freeing resources 318
 - show routines 321
 - task events register 320
- `eventSend()` 320
- examples
 - mutual exclusion 300
- exception handling 287
 - C++ 125
 - and interrupts 327
 - signal handlers 288
 - task `tExcTask` 293
- exception handling, synchronous 23
- exceptions
 - vThreads 14
- execution model 9
- `exit()` 281
- external stimuli, handling in vThreads 18

F

- fault detection
 - health monitoring 213
- FIFO
 - message queues, Wind 310
 - POSIX 98

- floating-point
 - (PowerPC) 340
 - routines unavailable 341
- floating-point exceptions, support for in partitions 342
- floating-point support
 - interrupt service routine limitations 326
 - task options 279
- fno-exceptions compiler option (C++) 125
- fno-rtti compiler option (C++) 126
- `fpExcEnable` parameter 342
- `fppArchLib` 326
- `FPSCR` 342
- `free()` 326

G

- global variables 290
- GNU compiler
 - configuring 331
 - CPU type, defining 330
- guard pages (vThreads) 31
 - defaults 31
 - limitations 32

H

- hardware
 - interrupts, *see* interrupt service routines
- hash table, resizing and relocating 336
- health monitoring
 - actions 210
 - alarm escalation 210
 - alarm injection 204
 - alarms 196
 - basic concepts 196
 - callback 212
 - configuration tables 215
 - dispatching 205
 - event reformatting 208
 - events 196
 - fault detection 213

- for COIL partitions
 - health monitoring 217
- hierarchy 197
- injection 202
- introduction 195
- logging 211
- messages 196
- module mode (HM_MODULE_MODE) 201
- notification 212
- partition mode
 - (HM_PARTITION_MODE) 202
- process mode (HM_PROCESS_MODE) 202
- public information 218
- thresholds 209
- heaps
 - cache 340
 - kernel 340
- HI macro 339
- HIADJ macro 339
- HM_MODULE_MODE 201
- HM_PARTITION_MODE 202
- HM_PROCESS_MODE 202
- hooks, task
 - routines callable by 285

I

- I/O
 - application multiplexed I/O 256
 - vThreads 14
- I/O permission bitmasks 138
- I/O system
 - asynchronous I/O
 - aioPxLib 90
- INCLUDE_POSIX_SIGNALS 121
- INCLUDE_SIGNALS 316
- initialization
 - user-supplied code in system shared
 - libraries 143
 - vThreads 25
- injection
 - health monitoring 202
- instruction MMU 332

- intConnect()
 - write protection, changing 152
- intCount() 323
- inter-module communication 74
- interrupt handling
 - application code, connecting to
 - callable routines 323
 - and exceptions 327
 - hardware, *see* interrupt service routines
 - stacks 324
- interrupt latency 295
- interrupt levels 327
- interrupt masking 327
- interrupt service routines (ISR) 323
 - see also* interrupt handling; interrupts;
 - intArchLib(1); 323
 - limitations 324
 - logging 326
 - see also* logLib(1)
 - and message queues 328
 - and pipes 328
 - routines callable from 324
 - and semaphores 328
 - and signals 316, 328
- interrupt stacks 324
- interrupts
 - locking 295
 - task-level code, communicating to 328
 - vThreads 14
- intertask communications 294, 316
- intLevelSet() 323
- intLock() 323
- intLockLevelSet() 327
- intUnlock() 323
- intVecBaseGet() 323
- intVecBaseSet() 323
- intVecGet() 323
- intVecSet() 323
- iostreams (C++) 127
- ISR, *see* interrupt service routines

K

- kernel
 - see also* Wind facilities
 - and multitasking 270
 - POSIX and Wind features, comparison of 89
 - message queues 109, 110
 - scheduling 98
 - semaphores 102
 - priority levels 273
- kernel heap 340
- kernelTimeSlice() 273, 274
- kill() 120, 316, 317
- killing
 - tasks 281

L

- latency
 - interrupt locks 295
 - preemptive locks 296
- loading 8
- loading and booting 8
- local ports 70
- locking
 - interrupts 295
 - semaphores 101
 - task preemptive locks 276, 296
- logging
 - health monitoring 211
- logging facilities
 - and interrupt service routines 326
 - task tLogTask 293
- longjmp() 288

M

- malloc()
 - interrupt service routine limitations 326
- memAttrAlloc() 149
- memAttrFree() 149
- memAttrWrite() 149

- memory
 - locking (POSIX) 91
 - pool 290
 - vThreads 14
- memory layout (PowerPC) 343
- memory management
 - vThreads 24
- memory partitions 148
 - see also* memAttrLib; memPartBaseLib; memPartLib
 - access permissions, working with 149
- message payload 200
- message queue permission bitmasks 139
- message queues 309
 - see also* msgQLib(1)
 - and VxWorks events 313
 - client-server example 312
 - displaying attributes 112, 312
 - and interrupt service routines 328
 - POSIX 109
 - see also* mqPxLib(1)
 - attributes 110, 112
 - code examples
 - attributes, examining 110, 112
 - checking for waiting message 116, 120
 - communicating by message queue 113, 115
 - notifying tasks 115
 - unlinking 113
 - Wind facilities, differences from 109, 110
 - priority setting 311
 - Wind 310, 312
 - code example 311
 - creating 310
 - deleting 310
 - queueing order 310
 - receiving messages 310
 - sending messages 310
 - timing out 310
 - waiting tasks 310
- messages
 - broadcast 68
 - directed 68
 - health monitoring 196

MMU 332
 attributes, access 150
 data 332
 instruction 332
 MMU_ATTR_CACHE_COPYBACK 151
 MMU_ATTR_CACHE_DEFAULT 151
 MMU_ATTR_CACHE_IO 151
 MMU_ATTR_CACHE_OFF 151
 MMU_ATTR_CACHE_WRITETHRU 151
 MMU_ATTR_PROT_SUP_EXE 151
 MMU_ATTR_PROT_SUP_READ 151
 MMU_ATTR_PROT_SUP_WRITE 151
 MMU_ATTR_PROT_USR_EXE 151
 MMU_ATTR_PROT_USR_READ 151
 MMU_ATTR_PROT_USR_WRITE 151
 MMU_ATTR_SPL_0-7 151
 module mode (HM_MODULE_MODE)
 health monitoring 201
 mq_close() 113
 mq_getattr() 110
 mq_notify() 115, 120
 mq_open() 112
 mq_receive() 113
 mq_send() 113
 mq_setattr() 110
 mq_unlink() 113
 mqPxLib 109
 msgQCreate() 310
 msgQDelete() 310
 msgQReceive() 310
 msgQSend() 310, 321
 MSR 342
 multitasking 270, 288
 example 292
 mutexes (POSIX) 108
 mutual exclusion 295
 see also semLib(1)
 code example 300
 counting semaphores 305
 interrupt locks 295
 preemptive locks 296
 and reentrancy 290

Wind semaphores 302, 305
 binary 300
 deletion safety 304
 priority inheritance 303
 priority inversion 302
 recursive use 304

N

named semaphores (POSIX) 101
 using 105
 nanosleep() 282, 283
 using 91
 notification
 health monitoring 212
 NULL pointer dereference detection 337
 null ports 70

O

O_NONBLOCK 110
 O_CREAT 105
 O_EXCL 105
 O_NONBLOCK 113
 OE exception 342
 online-loaded partitions 165

P

PAGE_MGR_ATTR_ALLOC_CONTIG 154
 PAGE_MGR_ATTR_ALLOC_MAPPED 154
 PAGE_MGR_ATTR_ALLOC_NONCONTIG 154
 PAGE_MGR_ATTR_ALLOC_UNMAPPED 154
 page-oriented memory 153
 see online mmanPxLib; pgMgrLib; pgPoolLib;
 pgPoolLstLib; rgnLib
 partition activation
 TPS scheduling 170
 partition configuration record 135

- partition I/O
 - COIL, *see* COIL I/O 264
 - vThreads, *see* vThreads I/O 221
- partition mode (HM_PARTITION_MODE)
 - health monitoring 202
- partition restart and device drivers (vThreads) 30
- partitionLib 165
- partition-related components
 - core OS 165
- partition-related routines
 - core OS 165
- partitions 135
 - cold start/restart 160
 - floating-point exceptions, support for 342
 - partition-related components in core OS 165
 - partition-related routines in core OS 165
 - scheduling by core OS 169
 - support in core OS 165
 - TPS scheduling 170
 - warm restart 161
- partition-safe text I/O
 - see* application multiplexed I/O
- partitionShow 165
- pause() 317
- payload images
 - RAM 158
 - ROM 157
- payloadLib 165
- pending tasks 271
- pending-suspended tasks 271
- persistent data
 - how handled 164
 - limitation 164
 - specifying 163
 - support for restart 163
- pgMgrPageAlloc() 154, 155
- pgMgrPageAllocAt() 154, 155
- pipeDevCreate() 315
- pipes 315
 - interrupt service routines 328
 - select(), using with 315
- port mapping 74
- port permission bitmasks 139
- ports
 - local ports 70
 - null ports 70
 - pseudo-ports 70
 - queuing ports 69
 - refresh rate 69
 - sampling ports 69
 - SAP ports 187
- POSIX
 - clocks 90, 91
 - see also* clockLib(1)
 - and kernel 89, 90
 - memory-locking interface 91
 - message queues 109
 - see also* message queues; mqPxLib(1)
 - mutex attributes 108
 - prioceling attribute 108
 - protocol attribute 108
 - priority limits, getting task 101
 - priority numbering 98
 - scheduling 97
 - see also* scheduling; 97
 - semaphores 101, 107
 - see also* semaphores; semPxLib(1)
 - signal functions 120
 - see also* signals; sigLib(1)
 - routines 317
 - task priority, setting 98, 100
 - code example 99
 - thread attributes 92
 - contentionscope attribute 93
 - detachstate attribute 93
 - inheritsched attribute 94
 - schedparam attribute 95
 - schedpolicy attribute 94
 - specifying 95
 - stackaddr attribute 92
 - stacksize attribute 92
 - threads 92
 - timers 90
 - see also* timerLib(1)
- Wind features, differences from 90
 - message queues 109
 - scheduling 98
 - semaphores 102

- posixPriorityNumbering global variable 98
- PowerPC 604 core 343
- PowerPC considerations 329
- PPC 60x
 - BAT model 332
 - memory mapping 332
 - segment model 332
- PPS
 - scheduling bitmasks 139
 - scheduling parameters 140
- PPS scheduling 17, 172
- PPS scheduling bitmasks 139
- preemptive locks 276, 296
- preemptive priority scheduling 100, 274
- printErrno() 287
- prioceiling attribute 108
- priority
 - inheritance 303
 - inversion 302
 - message queues 311
 - numbering 98
 - preemptive, scheduling 100, 274
 - task, setting
 - POSIX 98
 - Wind 273
- priority-preemptive scheduling 17
- process mode (HM_PROCESS_MODE)
 - health monitoring 202
- processes (POSIX) 98
- processor mode 338
- protection domains
 - page-oriented memory 153
 - virtual contexts 152
- protocol attribute 108
- pseudo-interrupts
 - events
 - forbidden in user handlers 21
 - permitted in user handlers 22
 - signals 19
- pseudo-ports 70, 185
- pthread_attr_getdetachstate() 93
- pthread_attr_getinheritsched() 94
- pthread_attr_getschedparam() 95
- pthread_attr_getscope() 93
- pthread_attr_getstackaddr() 92

- pthread_attr_getstacksize() 92
- pthread_attr_setdetachstate() 93
- pthread_attr_setinheritsched() 94
- pthread_attr_setschedparam() 95
- pthread_attr_setscope() 93
- pthread_attr_setstackaddr() 92
- pthread_attr_setstacksize() 92
- pthread_attr_t 92
- pthread_cleanup_pop() 97
- pthread_cleanup_push() 97
- PTHREAD_CREATE_DETACHED 93
- PTHREAD_CREATE_JOINABLE 93
- PTHREAD_EXPLICIT_SCHED 94
- pthread_getschedparam() 95
- pthread_getspecific() 96
- PTHREAD_INHERIT_SCHED 94
- pthread_key_create() 96
- pthread_key_delete() 96
- pthread_mutex_getprioceiling() 109
- pthread_mutex_setprioceiling() 109
- pthread_mutexattr_getprioceiling() 108
- pthread_mutexattr_getprotocol() 108
- pthread_mutexattr_setprioceiling() 108
- pthread_mutexattr_setprotocol() 108
- pthread_mutexattr_t 108
- PTHREAD_PRIO_INHERIT 108
- PTHREAD_PRIO_PROTECT 108
- PTHREAD_SCOPE_PROCESS 93
- PTHREAD_SCOPE_SYSTEM 93
- pthread_setcancelstate() 97
- pthread_setcanceltype() 97
- pthread_setschedparam() 95
- pthread_setspecific() 96
- pure code 289

Q

- queued signals 120
- queues
 - see also* message queues
 - ordering (FIFO vs. priority) 307
 - semaphore wait 307
- queuing ports 69

R

- raise() 317
- RAM payload images 158
- ready tasks 271
- reentrancy 289
- refresh rate 69
- register usage 338
- restart
 - cold
 - system 157, 160
 - implications for drivers 162
 - implications for I/O 163
 - vThreads 25
 - warm
 - partitions 161
 - system 159
- restart, vThreads 27
- ring buffers 326, 328
- ROM payload images 157
- root task tUsrRoot 293
- round-robin scheduling 18
 - defined 274
 - using 100, 101
- routines
 - scheduling, for 97
- rtasks
 - spawning 292
- run-time
 - execution model 9
 - layers 5
 - system 4
- Run-Time Type Information (RTTI) 126

S

- sampling ports 69
- SAP ports 187
- SCHED_FIFO 100
- sched_get_priority_max() 101
- sched_get_priority_min() 101
- sched_getparam()
 - scheduling parameters, describing 95
- sched_getscheduler() 100

- SCHED_RR 100
- sched_rr_get_interval() 101
- sched_setparam() 100
 - scheduling parameters, describing 95
- sched_setscheduler() 99
- schedPxLib 97, 98
- scheduler permission bitmask 139
- scheduling 273, 276
 - POSIX 97, 101
 - see also* schedPxLib(1) 97
 - algorithms 98
 - code example 100
 - FIFO 98, 100
 - policy, displaying current 100
 - preemptive priority 100
 - priority limits 101
 - priority numbering 98
 - round-robin 100, 101
 - routines for 97
 - time slicing 101
 - Wind facilities, differences from 98
 - Wind
 - preemptive locks 276, 296
 - preemptive priority 274
 - round-robin 274, 275
- scheduling (vThreads) 16
- scheduling rules
 - TPS scheduling 170
- segment model (PPC 60x) 332
- select()
 - and pipes 315
- sem_close() 106
- SEM_DELETE_SAFE 304
- sem_init() 103
- SEM_INVERSION_SAFE 303
- sem_open() 105
- sem_unlink() 106
- semaphores 107, 296
 - and VxWorks events 307
 - see also* semLib(1)
 - counting 102
 - example 305
 - deleting 102, 298
 - giving and taking 101, 298
 - and interrupt service routines 328, 326

- locking 101
- POSIX 101, 107
 - see also* semPxBLib(1)
 - named 101, 105, 107
 - code example 106
 - unnamed 101, 102, 103, 105
 - code example 103
 - Wind facilities, differences from 102
- posting 101
- recursive 304
 - code example 304
- synchronization 297, 305
 - code example 300, 301
- unlocking 101
- waiting 101
- Wind 297, 309
 - binary 298
 - code example 300
 - control 297
 - counting 305
 - mutual exclusion 300, 302
 - queuing 307
 - synchronization 300, 301
 - timing out 306
- semBCreate() 297
- semCCreate() 297
- semDelete() 297
- semFlush() 297, 302
- semGive() 298, 321
- semMCreate() 298
- semPxBLib 102
- semTake() 298
- service access point ports 187
- set_terminate() (C++) 126
- setjmp() 288
- shared code 288
- shared data regions 143
 - configuration structure 144
- shared data structures 294, 295
- shared libraries 142
 - support 343
- show() 312
- show() 112
- sigaction() 120, 316, 317
- sigaddset() 317
- sigblock() 316, 317
- sigdelset() 317
- sigemptyset() 317
- sigfillset() 317
- sigInit() 316
- sigismember() 317
- sigmask() 317
- signal handlers 316
- signal() 317
- signals 316
 - see also* sigLib(1)
 - configuring 316
 - and interrupt service routines 316, 328
 - POSIX 120, 121
 - queued 120
 - routines 317
 - signal handlers 316
 - UNIX BSD 316
 - routines 317
- sigpending() 317
- sigprocmask() 316, 317
- sigqueue() 120
 - buffers to, allocating 121
- sigqueueInit() 121
- sigsetmask() 316, 317
- sigsuspend() 317
- sigtimedwait() 121
- sigvec() 316, 317
- sigwaitinfo() 121
- spare-time monitoring
 - TPS scheduling 171
- spawning tasks 277, 292
- sslMain.c 143
- stack overflow protection (vThreads) 30
- stacks
 - interrupt 324
 - no fill 279
 - task exception 141
- Standard Template library (STL) 128
- start/restart
 - cold
 - system 157, 160
- suspended tasks 271

- synchronization (task) 297
 - code example 300, 301
 - counting semaphores, using 305
 - semaphores 300, 301
- system call permission bitmasks 138
 - custom permissions 140
 - I/O permissions 138
 - message queue permissions 139
 - port permissions 139
 - PPS scheduling 139
 - scheduler permissions 139
- system call stacks 141
- system calls, for vThreads 34
- system clock 283
- system cold start/restart 157
- system shared library, init routines for user-supplied
 - code 143
- system tasks 292
- system time 169
- system warm restart 159

T

- target shell
 - task tShell 293
- task control blocks (TCB) 270, 283, 291, 324
- task exception stacks 141
- TASK_EXC_STACK_SIZE 141
- taskActivate() 277
- taskCreateHookAdd() 283
- taskCreateHookDelete() 283
- taskDelay() 282
- taskDelete() 281
- taskDeleteHookAdd() 284
- taskDeleteHookDelete() 284
- taskIdListGet() 280
- taskIdSelf() 278
- taskIdVerify() 278
- taskInfoGet() 280
- taskInit() 277
- taskIsReady() 280
- taskIsSuspended() 280
- taskLock() 273
- taskName() 278
- taskNameToId() 278
- taskOptionsGet() 280
- taskOptionsSet() 280
- taskPriorityGet() 280
- taskPrioritySet() 273
- taskRegsSet() 280
- taskRestart() 282
- taskResume() 282
- tasks
 - blocked 276
 - contexts 270
 - creating 277
 - control blocks 270, 283, 291, 324
 - creating 277
 - delayed 271
 - delayed-suspended 271
 - delaying 271, 282, 322, 323
 - deleting safely 280
 - code example 282
 - semaphores, using 304
 - displaying information about 280
 - error status values 285
 - see also* errnoLib(1)
 - exception handling 287, 288
 - see also* signals; sigLib(1); excLib(1)
 - tExcTask 293
 - executing 282
 - hooks
 - see also* taskHookLib(1)
 - extending with 283
 - troubleshooting 284
 - IDs 278
 - interrupt level, communicating at 328
 - logging (tLogTask) 293
 - names 278
 - automatic 278
 - option parameters 279
 - pended 271
 - pended-suspended 271
 - priority, setting
 - driver support tasks 276
 - POSIX 98, 100
 - code example 99
 - Wind 273
 - ready 271

- root (tUsrRoot) 293
- scheduling
 - POSIX 97, 101
 - preemptive locks 276, 296
 - preemptive priority 100, 274
 - priority limits, getting 101
 - round-robin 274
 - see also* round-robin scheduling
 - time slicing 101
 - Wind 273
- shared code 288
- and signals 288, 316
- spawning 277, 292
- stack allocation 278
- states 271
- suspended 271
- suspending and resuming 282
- synchronization 297
 - code example 300, 301
 - counting semaphores, using 305
- system 292
- target shell (tShell) 293
- task events register 320
 - API 320
- telnet (tTelnetd, tTelnetInTask, tTelnetOutTask) 293
- variables 291
 - see also* taskVarLib(1)
 - context switching 291
- taskSafe() 281
- taskSpawn() 277
- taskStatusString() 280
- taskSuspend() 282
- taskSwitchHookAdd() 283
- taskSwitchHookDelete() 284
- taskTcb() 280
- taskUnlock() 273
- taskUnsafe() 281
- taskVarAdd() 291
- taskVarDelete() 291
- taskVarGet() 291
- taskVarSet() 291
- telnet
 - daemon tTelnetd 293
- terminate() (C++) 126

- threads 14
- threads (POSIX) 92
 - attributes 92, 96
 - specifying 95
 - keys 96
 - private data, accessing 96
 - terminating 97
- thresholds
 - health monitoring 209
- time management
 - vThreads 15
- time slicing 274
 - determining interval length 101
- timeout
 - message queues 310
 - semaphores 306
- timeouts
 - semaphores 306
- timer queue (vThreads) 15
- timers
 - see also* timerLib(1)
 - message queues, for (Wind) 310
 - POSIX 90, 91
 - semaphores, for (Wind) 306
 - watchdog 322, 323
 - code examples 322
- TPS scheduling 170
 - mode-based scheduling 171
 - partition activation 170
 - scheduling rules 170
 - spare-time monitoring 171
- trusted partitions 212

U

- unnamed semaphores (POSIX) 101, 102, 103
- user configuration record regions 147

V

variables

- global 290
- static data 290
- task 291

VE exception 342

virtual memory

- see also* vmLib
- attributes, MMU access 150

virtual memory contexts

- protection domains, and 152

vmPgAttrSet() 152

VT_EVENT_CLOCK_TICK 21

VT_EVENT_PORT_INT_RECV 22

VT_EVENT_PORT_INT_SEND 22

VT_EVENT_RELEASE_POINT 22

VT_EVENT_SC_COMPLETE 21

VT_EVENT_SYNC 21

VT_EVENT_USER 22

VT_EVENT_WARM_RESART 21

vThreads

- APIs 33
- APPS scheduling and 176
- boot sequence 25
- certifiability 7
- certification 15
- COLD versus WARM restarts 27
- cooperative WARM partition restart mechanism 28
- device I/O 33
- exception handling, synchronous 23
- exceptions 14
- external stimuli, handling 18
- guard pages 31
 - defaults 31
 - limitations 32
- I/O 14
- initialization 25
- interrupts 14
- key features 3
- memory 14
- memory management 24
- overview 13
- partition restart and device drivers 30

priority-preemptive scheduling 17

pseudo-interrupts

- events
 - forbidden in user handlers 21
 - permitted in user handlers 22
- signals 19

restart 25, 27

round-robin scheduling 18

scheduling 16

stack overflow protection 30

system call complete 21

system calls 34

system clock ticks 21

threads 14

time management 15

timer queue 15

vThreads I/O 221

device driver models 223

- core OS level 225

- split level 226

- vThreads level 224

sample drivers, communicating using ARINC

- ports 229

select() 228

worker tasks 222

vThreads Layer 7

VX_ALTIVEC_TASK 279

VX_DSP_TASK 279

VX_FP_TASK 279

VX_FP_TASK 125

VX_FP_TASK option 279

VX_NO_STACK_FILL 279

VX_PRIVATE_ENV 279

VX_UNBREAKABLE 279

VxWorks 653

- DO-178B certifiability 9

VxWorks 653

- overview 2

W

- WAIT_FOREVER 306
- warm restart
 - partitions 161
 - system 159
- watchdog timers 322
 - code examples
 - creating a timer 322
- wdCancel() 322
- wdCreate() 322
- wdDelete() 322
- wdStart() 322
- Wind facilities 90
 - message queues 310
 - POSIX, differences from 90
 - message queues 109, 110
 - scheduling 98
 - semaphores 102
 - scheduling 273, 276
 - semaphores 297
- wind kernel, *see* kernel
- worker tasks 168
 - vThreads I/O 222
- workQPanic 327
- write protection
 - exception vector tables, of 152

Z

- ZE exception 342