

WIND RIVER

VxWorks®

KERNEL API REFERENCE
Volume 1: Libraries

6.6

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, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark 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 94501-1153
U.S.A.

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

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

<http://www.windriver.com>

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

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

Contents

The *VxWorks Kernel API Reference* is a two-volume set that provides reference entries describing the facilities available in the VxWorks kernel. For reference entries that describe facilities for VxWorks process-based application development, see the *VxWorks Application API Reference*. For reference entries that describe VxWorks drivers, see the *VxWorks Drivers API Reference*.

Volume 1: Libraries

Volume 1 (this book) provides reference entries for each of the VxWorks kernel libraries, arranged alphabetically. Each entry lists the routines found in the library, including a one-line synopsis of each and a general description of their use.

Individual reference entries for each of the available functions in these libraries is provided in Volume 2.

Volume 2: Routines

Volume 2 provides reference entries for each of the routines found in the VxWorks kernel libraries documented in Volume 1.

Volume **1**

Libraries

adrSpaceLib – address space allocator	9
adrSpaceShow – address space show library	10
aimCacheLib – cache library of the Architecture Independent Manager	10
aimFppLib – floating-point unit support library for AIM	13
aimMmuLib – MMU Architecture Independent Manager	13
aioPxLib – asynchronous I/O (AIO) library (POSIX)	14
aioPxShow – asynchronous I/O (AIO) show library	18
aioSysDrv – AIO system driver	18
am79c97xVxbEnd – AMD Am79c97x PCnet/PCI VxBus END driver	18
an983VxbEnd – Infineon AN983B/BX VxBus END driver	19
bLib – buffer manipulation library	20
bcm52xxPhy – driver for Broadcom bcm52xx 10/100 ethernet PHY chips	21
bootInit – ROM initialization module	21
bootLib – boot ROM subroutine library	23
bootParseLib – boot ROM bootline interpreter library	24
cacheArchLib – architecture-specific cache management library	25
cacheAuLib – Alchemy Au cache management library	26
cacheLib – cache management library	26
cacheR10kLib – MIPS R10000 cache management library	35
cacheR4kLib – MIPS R4000 cache management library	35
cacheR5kLib – MIPS R5000 cache management library	36
cacheR7kLib – MIPS R7000 cache management library	36
cacheSh7750Lib – Renesas SH7750 cache management library	37
cacheTx49Lib – Toshiba Tx49 cache management library	37
cbioLib – Cached Block I/O library	38
cdromFsLib – ISO 9660 CD-ROM read-only file system library	42
clockLib – clock library (POSIX)	46
cnsCompLib – Media type comp library	47
cnsLib – Component notification system library	47
coreDumpHookLib – core dump hook library	49

coreDumpLib – core dump library 49
coreDumpMemFilterLib – core dump memory filtering library 53
coreDumpShow – core dump show routines 54
coreDumpUtilLib – core dump utility library 54
cplusLib – basic run-time support for C++ 55
cpuPwrLightLib – light power manager library (x86, PPC and VxSim) 56
cpuPwrUtilLib – utilization-based CPU power manager (x86 only) 58
cpuset – cpuset_t type manipulation macros 59
dbgArchLib – architecture-dependent debugger library 59
dbgLib – shell debugging facilities 61
dcacheCbio – Disk Cache Driver 63
dirLib – directory handling library (POSIX) 67
dosFsCacheLib – MS-DOS media-compatible Cache library 69
dosFsFmtLib – MS-DOS media-compatible file system formatting library 71
dosFsLib – MS-DOS media-compatible file system library 71
dosFsShow – DosFS Show routines 84
dpartCbio – generic disk partition manager 84
dshmMuxLib – DSHM service/hardware bus multiplexer 85
dsiSockLib – DSI sockets library 88
edrErrLogLib – the ED&R error log library 88
edrLib – Error Detection and Reporting subsystem 90
edrShow – ED&R Show Routines 92
edrSysDbgLib – ED&R system-debug flag 92
envLib – environment variable library 93
errnoLib – error status library 94
eventLib – VxWorks events library 96
excArchLib – architecture-specific exception-handling facilities 100
excLib – generic exception handling facilities 100
fccVxbEnd – fcc VxBus END driver 101
fecVxbEnd – Motorola/Freescale FEC VxBus END driver 102
fei8255VxbEnd – Intel PRO/100 VxBus END driver 104
ffsLib – find first bit set library 106
fioBaseLib – formatted I/O library 106
fioLib – formatted I/O library 107
fppArchLib – architecture-dependent floating-point coprocessor support 108
fppLib – floating-point coprocessor support library 112
fppShow – floating-point show routines 113
fsEventUtilLib – Event Utility functions for different file systems 114
fsMonitor – The File System Monitor 114
fsPxLib – I/O, file system API library (POSIX) 114
fttruncate – POSIX file truncation 115
gei825xxVxbEnd – Intel PRO/1000 VxBus END driver 115
getopt – getopt facility 117
hashLib – generic hashing library 117
hookLib – generic hook library for VxWorks 120

hookShow – hook show routines 121
hrFsLib – highly reliable file system library 122
hrFsTimeLib – time routines for HRFS 122
hrfsChkDskLib – HRFS Check disk library - Readonly version 122
hrfsFormatLib – HRFS format library 123
inflateLib – inflate code using public domain zlib functions 123
intArchLib – architecture-dependent interrupt library 127
intLib – architecture-independent interrupt subroutine library 129
ioLib – I/O interface library 130
iosLib – I/O system library 131
iosShow – I/O system show routines 132
isrLib – isr objects library 132
isrShow – isr objects show library 135
kern_sysctl – sysctl kernel routines 136
kernelLib – VxWorks kernel library 136
ledLib – line-editing library 138
loadLib – generic object module loader 141
logLib – message logging library 143
loginLib – user login/password subroutine library 144
lstLib – doubly linked list subroutine library 146
m85xxCCSR – VxBus driver for PowerPC 85xx CCSR resource allocation 147
mathALib – C interface library to high-level math functions 147
memDrv – pseudo memory device driver 149
memEdrLib – memory manager error detection and reporting library 151
memEdrRtpShow – memory error detection show routines for RTPs 154
memEdrShow – memory error detection show routines 154
memInfo – memory partition info routines 154
memLib – full-featured memory partition manager 155
memPartLib – core memory partition manager 157
memShow – memory show routines 159
miiBus – MII bus controller and API library 159
mmanPxLib – memory management library (POSIX) 161
mmanShow – mmap manager show library 161
mmuMapLib – MMU mapping library for ARM Ltd. processors 161
mmuPro32Lib – MMU library for Pentium II 162
mmuPro36Lib – MMU library for PentiumPro/2/3/4 36 bit mode 164
mmuShLib – Memory Management Unit Library for Renesas SH77xx 168
moduleLib – code module list management library 168
mountd – Mount protocol library 170
mqPxLib – message queue library (POSIX) 171
mqPxShow – POSIX message queue show 172
msgQEvLib – VxWorks events support for message queues 173
msgQInfo – message queue information routines 173
msgQLib – message queue library 174
msgQOpen – extended message queue library 176

msgQShow	– message queue show routines	177
msgQSmLib	– shared memory message queue library (VxMP Option)	177
mvYukonIIVxbEnd	– Marvell Yukon II VxBus END driver	178
mvYukonVxbEnd	– Marvell Yukon I VxBus END driver	180
ne2000VxbEnd	– NE2000 Compatible VxBus END driver	181
nfsCommon	– Network File System (NFS) I/O driver	183
nfsHash	– file based hash table for file handle to file name and reverse	184
nfdsd	– NFS Server Init routines	184
nfdsdCommon	– Common functions for v2 and v3	185
ns8381xVxbEnd	– National Semiconductor DP83815/6 VxBus END driver	186
ns83902VxbEnd	– NatSemi DP83902A ST-NIC VxBus END driver	187
objLib	– generic object management library	188
objShow	– wind objects show library	189
partLib	– routines to create disk partitions on a rawFS	189
passFsLib	– pass-through file system library (VxSim)	190
pentiumALib	– P5, P6 and P7 family processor specific routines	192
pentiumLib	– Pentium and Pentium[234] library	197
pentiumShow	– Pentium and Pentium[234] specific show routines	200
phil	– VxWorks/SMP Dijkstra's dining philosophers demo	201
pipeDrv	– pipe I/O driver	203
pmLib	– persistent memory library	205
poolLib	– Memory Pool Library	207
poolShow	– Wind Memory Pool Show Library	208
primesDemo	– VxWorks SMP prime number computation demo	208
pthreadLib	– POSIX 1003.1c thread library interfaces	212
ptyDrv	– pseudo-terminal driver	219
quiccEngineUtils	– quicc engine resource allocation	220
rBuffLib	– dynamic ring buffer (rBuff) library	220
ramDiskCbio	– RAM Disk Cached Block Driver	220
ramDrv	– RAM disk driver	221
rawFsLib	– raw block device file system library	222
rawPerfDemo	– VxWorks/SMP raw performance demo	224
rebootLib	– reboot support library	226
rngLib	– ring buffer subroutine library	226
rtl8139VxbEnd	– RealTek 8139/8100 10/100 VxBus END driver	227
rtl8169VxbEnd	– RealTek 8139C+/8101E/816x/811x VxBus Ethernet driver	228
rtpHookLib	– RTP Hook Support library	230
rtpLib	– Real Time Process library	231
rtpShow	– Real Time Process show routine	238
rtpSigLib	– RTP software signal facility library	239
rtpUtilLib	– Real Time Process Utility library	239
salClient	– socket application client library	240
salServer	– socket application server library	242
sbeVxbEnd	– Broadcom/Sibyte BCM1250 VxBus END driver	246
scMemVal	– helper routines to validate system call parameters	248

schedPxLib – scheduling library (POSIX) 248
scsi1Lib – Small Computer System Interface (SCSI) library (SCSI-1) 249
scsi2Lib – Small Computer System Interface (SCSI) library (SCSI-2) 253
scsiCommonLib – SCSI library common commands for all devices (SCSI-2) 259
scsiCtrlLib – SCSI thread-level controller library (SCSI-2) 260
scsiDirectLib – SCSI library for direct access devices (SCSI-2) 260
scsiLib – Small Computer System Interface (SCSI) library 261
scsiMgrLib – SCSI manager library (SCSI-2) 262
scsiSeqLib – SCSI sequential access device library (SCSI-2) 263
sdLib – shared data API layer 265
sdShow – Shared Data region show routine 267
selectLib – UNIX BSD select library 268
semBLib – binary semaphore library 269
semCLib – counting semaphore library 271
semEvLib – VxWorks events support for semaphores 273
semExchange – semaphore exchange library 273
semInfo – semaphore information routines 274
semLib – general semaphore library 275
semMLib – mutual-exclusion semaphore library 277
semOpen – extended semaphore library 280
semPxLib – semaphore synchronization library (POSIX) 280
semPxShow – POSIX semaphore show library 282
semRWLib – reader/writer semaphore library 282
semShow – semaphore show routines 284
semSmLib – shared memory semaphore library (VxMP Option) 285
shellConfigLib – the shell configuration management module 286
shellDataLib – the shell data management module 287
shellInterpCmdLib – the command interpreter library 288
shellInterpLib – the shell interpreters management module 289
shellLib – the kernel shell module 289
shellPromptLib – the shell prompt management module 292
shlShow – Shared Library Show Routine 293
sigLib – software signal facility library 294
smMemLib – shared memory management library (VxMP Option) 301
smMemShow – shared memory management show routines (VxMP Option) 303
smNameLib – shared memory objects name database library (VxMP Option) 304
smNameShow – shared memory objects name database show routines (VxMP Option) 306
smObjLib – shared memory objects library (VxMP Option) 307
smObjShow – shared memory objects show routines (VxMP Option) 309
smpLockDemo – synchronization mechanism demo for VxWorks SMP 310
snsLib – Socket Name Service library 312
snsShow – Socket Name Service show routines 315
spinLockLib – spinlock library 315
spyLib – spy CPU activity library 319
ssiDb – SSI database module 321

strSearchLib – Efficient string search library 321
symLib – symbol table subroutine library 322
symShow – symbol table show routines 325
sysLib – system-dependent library 325
syscallHookLib – SYSCALL Hook Support library 327
syscallLib – VxWorks System Call Infrastructure management library 328
syscallShow – VxWorks System Call Infrastructure management library 330
sysctl – sysctl command 331
tarLib – UNIX tar compatible library 331
taskArchLib – architecture-specific task management routines 332
taskHookLib – task hook library 332
taskHookShow – task hook show routines 334
taskInfo – task information library 334
taskLib – task management library 335
taskOpen – extended task management library 338
taskRotate – taskRotate functionality 339
taskShow – task show routines 339
taskUtilLib – task utility library 340
taskVarLib – task variables support library 341
tc3c905VxbEnd – 3Com 3c905/B/C VxBus END driver 341
tffsDrv – TrueFFS interface for VxWorks 343
tickLib – clock tick support library 346
timerLib – timer library (POSIX) 347
timerOpen – extended timer library 348
timerShow – POSIX timer show library 349
timexLib – execution timer facilities 349
tlsLib – thread local storage support library 350
trgLib – trigger events control library 352
trgShow – trigger show routine 352
tssecVxbEnd – Freescale TSEC VxBus END driver 353
ttyDrv – provide terminal device access to serial channels 354
tyLib – *tty* driver support library 355
unShow – information display routines for AF_LOCAL 360
unixDrv – UNIX-file disk driver (VxSim for Solaris) 360
unldLib – object module unloading library 362
usrConfig – user-defined system configuration library 364
usrFdiskPartLib – FDISK-style partition handler 364
usrFsLib – file system user interface subroutine library 366
usrLib – user interface subroutine library 368
usrRtpLib – Real Time Process user interface subroutine library 370
usrRtpStartup – RTP Startup Facility Support Code 370
usrShellHistLib – shell history user interface subroutine library 371
usrTransLib – Transaction Device Access Library 371
utfLib – Library to manage Unicode characters encoded in UTF-8 and UTF-16 372
virtualDiskLib – virtual disk driver library (vxSim) 372

vmArch32Lib – VM (VxVMI) library for PentiumPro/2/3/4 32 bit mode 374
vmArch36Lib – VM (VxVMI) library for PentiumPro/2/3/4 36 bit mode 375
vmBaseArch32Lib – VM (bundled) library for PentiumPro/2/3/4 32 bit mode 376
vmBaseArch36Lib – VM (bundled) library for PentiumPro/2/3/4 36 bit mode 376
vmBaseLib – base virtual memory support library 377
vmGlobalMap – virtual memory global mapping library 379
vmShow – virtual memory show routines 380
vrfsLib – the Virtual Root File System 380
vxAtomicLib – atomic operations library 381
vxCpuLib – CPU utility routines 383
vxLib – miscellaneous support routines 384
vxMemProbeLib – miscellaneous support routines 384
vxvEtsecEnd – Freescale Enhanced TSEC VxBus END driver 385
vxvFileNvRam – VxBus driver for NVRam on a filesystem file 390
vxvI8042Kbd – Intel 8042 keyboard driver routines 390
vxvIntelIchStorage – Intel ICH0/1 (82801) ATA/IDE and ATAPI CDROM 391
vxvIntelIchStorageShow – ICH ATA disk device driver show routine 397
vxvM6845Vga – motorola 6845 VGA console driver 398
vxvNonVolLib – non-volatile RAM to non-volatile memory routine mapping 398
vxvPcConsole – console handler 399
vxvSI31xxStorage – PCI bus header file for vxBus 399
vxvSmscLan9118End – SMSC LAN9118 VxBus END driver 401
vxvSimHostArchLib – VxSim host side interface library 403
wdLib – watchdog timer library 404
wdShow – watchdog show routines 405
wdbLib – WDB agent context management library 406
wdbMdlSymSyncLib – target-host modules and symbols synchronization 406
wdbUserEvtLib – WDB user event library 408
windPwrLib – Power Management Library 408
wvFileUploadPathLib – file destination for event data 409
wvLib – event logging control library (System Viewer) 409
wvSockUploadPathLib – socket upload path library 417
wvTmrLib – timer library (System Viewer) 417
wvTsfsUploadPathLib – target host connection library using TSFS 418
xbdBlkDev – XBD / BLK_DEV Converter 418
xbdCbioDev – XBD / CBIO Converter 419
xbdRamDisk – XBD Ramdisk Implementation 419
xbdTrans – Transaction extended-block-device 420

adrSpaceLib

NAME	adrSpaceLib – address space allocator
ROUTINES	adrSpacePageUnmap() – unmap a set of virtual pages adrSpaceRAMAddToPool() – add specified memory block to RAM pool adrSpaceRAMReserve() – reserve memory from the RAM pool adrSpaceVirtReserve() – reserve memory from the virtual space adrSpaceInfoGet() – get status of the address space library
DESCRIPTION	<p>The Address Space Allocator provides the functionality for managing virtual and physical RAM space for the kernel and user applications. It is used (and automatically included) when Real Time Process support (INCLUDE_RTP) is included in the kernel.</p> <p>The physical and address space management is initialized based on the BSPs memory configuration (LOCAL_MEM_LOCAL_ADRS, KERNEL_HEAP_SIZE, 'sysPhysMemDesc[]' and for PPC the sysBatDesc[]), and based on the the processor architecture's specific segmentation requirements (such as MIPS segments). During initialization, the following types of memory configuration errors are detected:</p> <ul style="list-style-type: none">- Virtual space overlaps between multiple sysPhysMemDesc[] entries.- Physical space overlaps between multiple sysPhysMemDesc[] entries.- Virtual or physical start address of sysPhysMemDesc[] entry not page aligned.- Length of sysPhysMemDesc[] entry not page aligned.- Entry in sysPhysMemDesc[] defines memory range in a user-only segment.- Entry in sysPhysMemDesc[] defines memory range that overflows either the physical or the virtual address space- Could not create all resources. This could happens if there is not enough memory in the kernel heap. <p>Note that in case of an overlap, the error will indicate the second of the overlapping entries, but not the first one.</p> <p>A special case is the configuration with MMU disabled. In this case the BSPs sysPhysMemDesc[] - if it exists - is ignored. In this case the address space managed by this library is restricted to the system RAM, allowing identity mapping only.</p> <p>The physical RAM pool is the collection of all RAM described in sysPhysMemDesc[] in the range LOCAL_MEM_LOCAL_ADRS to sysPhysMemTop(), managed with page size granularity. RAM memory outside of the range LOCAL_MEM_LOCAL_ADRS to sysPhysMemTop() can be added to the RAM pool by calling the routine adrSpaceRAMAddToPool().</p>
INCLUDE FILES	adrSpaceLib.h

SEE ALSO **adrSpaceShow**

adrSpaceShow

NAME **adrSpaceShow** – address space show library

ROUTINES **adrSpaceShow()** – display information about address spaces managed by **adrSpaceLib**

DESCRIPTION This library provides routines to display information about the physical and virtual address spaces managed by **adrSpaceLib**. This library is included whenever the component **INCLUDE_ADR_SPACE_SHOW** is added to the kernel configuration.

INCLUDE_FILES: **adrSpaceShow.h**

INCLUDE FILES none

SEE ALSO **adrSpaceLib**

aimCacheLib

NAME **aimCacheLib** – cache library of the Architecture Independent Manager

ROUTINES **aimCacheInit()** – initialize cache aim with supplied parameters

DESCRIPTION This library contains the support routines for the cache portion of the Architecture Independent Manager.

aimCacheInit()
 Is called by the bsp via an architecture specific initialization routine. It collects attribute information for all caches and publishes the attributes. It decides which AIM functions are to be called from the vxWorks API. It calculates maximum indices, counts, rounding factors, and so on, and creates local copies specific to the AIM routines in use.

aimCacheEnable()
 Calls the appropriate cache enable primitive (Icache, Dcache, etc) and maintains a local copy of the enabled state.

aimCacheDisable()
 Calls the appropriate cache disable primitive (Icache, Dcache, etc) and maintains a local copy of the enabled state.

aimCacheLock()

Rounds the address and count and calls the correct primitive. Optionally, it maintains a database of locked regions of cache, preventing invalidate commands from operating on locked regions.

If lock protection has been enabled by the specification of the **C_FLG_LOCKPRTKT** flag when the **cacheAimxxxLock** was declared in the **CACHECONFIG** structure, then the lock database is maintained by the cache AIM, identifying locked regions within cache.

This locked database is queried by the AIM prior to calling the invalidate or clear primitives. An error is returned if any locks are active in the described cache region.

Assumptions made using cache AIM database:

- (1) There can be no overlap/subsets of locked regions with regions to be invalidated or cleared; if so we return **ERROR**.
- (2) On a lock request, if the record is found to already exist within the cache AIM database, assume the region is already set; do nothing but return **OK**.
- (3) On an unlock request, if the record is found not to exist within cache AIM database, assume the region is not set; do nothing but return **OK**.

aimCacheUnlock()

Rounds the address and count and calls the correct primitive. Optionally, it can update a database of locked regions.

aimCacheIndexFlush()

Calculates the proper index and count and calls the flush primitive for the specified cache. In the case where the entire cache is specified, **size == ENTIRE_CACHE**, and a primitive to flush the entire cache exists, the index and count calculations are not required and the **xxxXcacheFlushAll** primitive is called.

aimCacheVirtFlush()

Rounds the virtual address and count as necessary and calls the flush primitive for the specified cache. In the case where the entire cache is specified, **size == ENTIRE_CACHE**, and a primitive to flush the entire cache exists, address and count rounding are not required and the **xxxXcacheFlushAll** primitive is called.

aimCachePhysFlush()

Rounds the virtual address and count as necessary, computes the correct physical address from the virtual address, and calls the flush primitive for the specified cache. In the case where the entire cache is specified, **size == ENTIRE_CACHE**, and a primitive to flush the entire cache exists, address and count rounding are not required and the **xxxXcacheFlushAll** primitive is called.

aimCacheIndexInvalidate()

Calculates the proper index and count and calls the invalidate primitive for the specified cache. In the case where the entire cache is specified, **size == ENTIRE_CACHE**, and a primitive to invalidate the entire cache exists, the index and count calculations are not required and the **xxxXcacheInvalidateAll** primitive is called.

aimCacheVirtInvalidate()

Rounds the virtual address and count as necessary and calls the invalidate primitive for the specified cache. In the case where the entire cache is specified, size == ENTIRE_CACHE, and a primitive to flush the entire cache exists, address and count rounding are not required and the xxxXcacheInvalidateAll primitive is called.

aimCachePhysInvalidate()

Rounds the virtual address and count as necessary, computes the correct physical address from the virtual address, and calls the invalidate primitive for the specified cache. In the case where the entire cache is specified, size == ENTIRE_CACHE, and a primitive to flush the entire cache exists, address and count rounding are not required and the xxxXcacheInvalidateAll primitive is called.

aimCacheIndexClear()

Calculates the proper index and count and calls the clear primitive for the specified cache. In the case where the entire cache is specified, size == ENTIRE_CACHE, and a primitive to invalidate the entire cache exists, the index and count calculations are not required and the xxxXcacheClearAll primitive is called. In the case where a Clear primitive does not exist, the appropriate Flush primitive is called, followed by a call to the Invalidate primitive.

aimCacheVirtClear()

Rounds the virtual address and count as necessary and calls the Clear primitive for the specified cache. In the case where the entire cache is specified, size == ENTIRE_CACHE, and a primitive to Clear the entire cache exists, address and count rounding are not required and the xxxXcacheClearAll primitive is called. In the case where a Clear primitive does not exist, the Flush primitive is called, followed by a call to the Invalidate primitive.

aimCachePhysClear()

Rounds the virtual address and count as necessary, computes the correct physical address from the virtual address, and calls the Clear primitive for the specified cache. In the case where the entire cache is specified, size == ENTIRE_CACHE, and a primitive to Clear the entire cache exists, address and count rounding are not required and the xxxXcacheClearAll primitive is called. In the case where a Clear primitive does not exist, the Flush primitive is called, followed by a call to the Invalidate primitive.

aimCacheTextUpdate()

This routine flushes the data cache, then invalidates the instruction cache. This operation forces the instruction cache to fetch code that may have been created via the data path. This is accomplished by calling the appropriate aimCacheFlush and aimCacheInvalidate routines.

There are no AIM-specific routines for the following functions. They map directly to the architecture-dependent primitive (if provided).

cacheDmaMalloc()

cacheDmaFree()

cacheDmaVirtToPhys()

cacheDmaPhysToVirt()

CONFIGURATION	The cache portion of the Architecture Independent Manager is automatically included in VxWorks when cache is enabled.
INCLUDE FILES	aimCacheLib.h, cacheLib.h

aimFppLib

NAME	aimFppLib – floating-point unit support library for AIM
ROUTINES	aimFppLibInit() – Initialize the AIM FPU library fppTaskRegsSet() – Sets FPU context for a task fppTaskRegsGet() – Gets FPU context for a task
DESCRIPTION	This is the Architecture Independant Manager (AIM) for VxWorks floating point support. It contains floating point routines that are common to all CPU architectures supported by VxWorks.
INCLUDE FILES	none
SEE ALSO	fppArchLib, coprocLib

aimMmuLib

NAME	aimMmuLib – MMU Architecture Independent Manager
ROUTINES	aimMmuLibInit() – initialize the AIM
DESCRIPTION	This library contains the Architecture Independent Manager (AIM) for the VxWorks MMU subsystem. This library creates generic structures for the AD-MMU to use in managing the hardware MMU. Because the structures are generic a lot of the code that manipulates the structures is generic. So the AIM is designed to do that manipulation for several different AD-MMUs. Specifically software TLB Miss MMUs. A lot of the complex code is now in this layer so the multiple AD-MMUs become simpler. HW restrictions mean some features supplied by the AIM are only available if the HW allows, such as variable size pages for dynamic TLBs and locking of regions of memory.

CONFIGURING VXWORKS

AIM for the VxWorks MMU subsystem is automatically included when the MMU is enabled.

INCLUDE FILES **vmLib.h, aimMmuLib.h**

aioPxLib

NAME **aioPxLib** – asynchronous I/O (AIO) library (POSIX)

ROUTINES **aio_read()** – initiate an asynchronous read (POSIX)
 aio_write() – initiate an asynchronous write (POSIX)
 lio_listio() – initiate a list of asynchronous I/O requests (POSIX)
 aio_suspend() – wait for asynchronous I/O request(s) (POSIX)
 aio_cancel() – cancel an asynchronous I/O request (POSIX)
 aio_fsync() – asynchronous file synchronization (POSIX)
 aio_error() – retrieve error status of asynchronous I/O operation (POSIX)
 aio_return() – retrieve return status of asynchronous I/O operation (POSIX)

DESCRIPTION This library implements asynchronous I/O (AIO) according to the definition given by the POSIX standard 1003.1b (formerly 1003.4, Draft 14). AIO provides the ability to overlap application processing and I/O operations initiated by the application. With AIO, a task can perform I/O simultaneously to a single file multiple times or to multiple files.

After an AIO operation has been initiated, the AIO proceeds in logical parallel with the processing done by the application. The effect of issuing an asynchronous I/O request is as if a separate thread of execution were performing the requested I/O.

AIO LIBRARY The AIO library is initialized by calling **aioPxLibInit()**, which should be called once (typically at system start-up) after the I/O system has already been initialized.

AIO COMMANDS The file to be accessed asynchronously is opened via the standard `open` call. `Open` returns a file descriptor which is used in subsequent AIO calls.

The caller initiates asynchronous I/O via one of the following routines:

- aio_read()**
initiates an asynchronous read
- aio_write()**
initiates an asynchronous write
- lio_listio()**
initiates a list of asynchronous I/O requests

Each of these routines has a return value and error value associated with it; however, these values indicate only whether the AIO request was successfully submitted (queued), not the ultimate success or failure of the AIO operation itself.

There are separate return and error values associated with the success or failure of the AIO operation itself. The error status can be retrieved using **`aio_error()`**; however, until the AIO operation completes, the error status will be **`EINPROGRESS`**. After the AIO operation completes, the return status can be retrieved with **`aio_return()`**.

The **`aio_cancel()`** call cancels a previously submitted AIO request. The **`aio_suspend()`** call waits for an AIO operation to complete.

Finally, the **`aioShow()`** call (not a standard POSIX function) displays outstanding AIO requests.

AIO CONTROL BLOCK

Each of the calls described above takes an AIO control block (**`aiocb`**) as an argument. The calling routine must allocate space for the **`aiocb`**, and this space must remain available for the duration of the AIO operation. (Thus the **`aiocb`** must not be created on the task's stack unless the calling routine will not return until after the AIO operation is complete and **`aio_return()`** has been called.) Each **`aiocb`** describes a single AIO operation. Therefore, simultaneous asynchronous I/O operations using the same **`aiocb`** are not valid and produce undefined results.

The **`aiocb`** structure and the data buffers referenced by it are used by the system to perform the AIO request. Therefore, once the **`aiocb`** has been submitted to the system, the application must not modify the **`aiocb`** structure until after a subsequent call to **`aio_return()`**. The **`aio_return()`** call retrieves the previously submitted AIO data structures from the system. After the **`aio_return()`** call, the calling application can modify the **`aiocb`**, free the memory it occupies, or reuse it for another AIO call.

As a result, if space for the **`aiocb`** is allocated off the stack the task should not be deleted (or complete running) until the **`aiocb`** has been retrieved from the system via an **`aio_return()`**.

The **`aiocb`** is defined in **`aio.h`**. It has the following elements:

```
struct
{
    int                aio_fildes;
    off_t              aio_offset;
    volatile void *    aio_buf;
    size_t              aio_nbytes;
    int                aio_reqprio;
    struct sigevent     aio_sigevent;
    int                aio_lio_opcode;
    AIO_SYS             aio_sys;
} aiocb
```

`aio_fildes`

file descriptor for I/O.

aio_offset

offset from the beginning of the file where the AIO takes place. Note that performing AIO on the file does not cause the offset location to automatically increase as in read and write; the caller must therefore keep track of the location of reads and writes made to the file and set **aio_offset** to correct value every time. AIO lib does not manage this offset for its applications.

aio_buf

address of the buffer from/to which AIO is requested.

aio_nbytes

number of bytes to read or write.

aio_reqprio

amount by which to lower the priority of an AIO request. Each AIO request is assigned a priority; this priority, based on the calling task's priority, indicates the desired order of execution relative to other AIO requests for the file. The **aio_reqprio** member allows the caller to lower (but not raise) the AIO operation priority by the specified value. Valid values for **aio_reqprio** are in the range of zero through **AIO_PRIO_DELTA_MAX**. If the value specified by **aio_req_prio** results in a priority lower than the lowest possible task priority, the lowest valid task priority is used.

aio_sigevent

(optional) if nonzero, the signal to return on completion of an operation.

aio_lio_opcode

operation to be performed by a **lio_listio()** call; valid entries include **LIO_READ**, **LIO_WRITE**, and **LIO_NOP**.

aio_sys

a Wind River Systems addition to the **aioCb** structure; it is used internally by the system and must not be modified by the user.

EXAMPLES

A writer could be implemented as follows:

```
if ((pAioWrite = calloc (1, sizeof (struct aioCb))) == NULL)
{
    printf ("calloc failed\n");
    return (ERROR);
}

pAioWrite->aio_fildes = fd;
pAioWrite->aio_buf = buffer;
pAioWrite->aio_offset = 0;
strcpy (pAioWrite->aio_buf, "test string");
pAioWrite->aio_nbytes = strlen ("test string");
pAioWrite->aio_sigevent.sigev_notify = SIGEV_NONE;

aio_write (pAioWrite);

/* .
.
```

```

do other work
.
.
    */

    /* now wait until I/O finishes */

    while (aio_error (pAioWrite) == EINPROGRESS)
        taskDelay (1);

    aio_return (pAioWrite);
    free (pAioWrite);

```

A reader could be implemented as follows:

```

/* initialize signal handler */

action1.sa_sigaction = sigHandler;
action1.sa_flags     = SA_SIGINFO;
sigemptyset(&action1.sa_mask);
sigaction (TEST_RT_SIG1, &action1, NULL);

if ((pAioRead = calloc (1, sizeof (struct aiocb))) == NULL)
{
    printf ("calloc failed\n");
    return (ERROR);
}

pAioRead->aio_fildes = fd;
pAioRead->aio_buf     = buffer;
pAioRead->aio_nbytes  = BUF_SIZE;
pAioRead->aio_sigevent.sigev_signo = TEST_RT_SIG1;
pAioRead->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
pAioRead->aio_sigevent.sigev_value.sival_ptr = (void *)pAioRead;

aio_read (pAioRead);

/*
.
.
    do other work
.
.
    */

```

The signal handler might look like the following:

```

void sigHandler
(
    int                sig,
    struct siginfo     info,
    void *             pContext
)
{
    struct aiocb *     pAioDone;

```

```
pAioDone = (struct aiocb *) info.si_value.sival_ptr;
aio_return (pAioDone);
free (pAioDone);
}
```

INCLUDE FILES	aio.h
SEE ALSO	POSIX 1003.1b document

aioPxShow

NAME	aioPxShow – asynchronous I/O (AIO) show library
ROUTINES	aioShow() – show AIO requests
DESCRIPTION	This library implements the show routine for aioPxLib .
INCLUDE FILES	aio.h

aioSysDrv

NAME	aioSysDrv – AIO system driver
ROUTINES	aioSysInit() – initialize the AIO system driver
DESCRIPTION	This library is the AIO system driver. The system driver implements asynchronous I/O with system AIO tasks performing the AIO requests in a synchronous manner. It is installed as the default driver for AIO.
INCLUDE FILES	aioSysDrv.h
SEE ALSO	POSIX 1003.1b document

am79c97xVxbEnd

NAME	am79c97xVxbEnd – AMD Am79c97x PCnet/PCI VxBus END driver
------	---

ROUTINES	InPciRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver for the AMD Am79C97x PCnet/PCI family of PCI 10/100 ethernet controllers. The Am79C97x family is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications.</p> <p>The PCnet/PCI family encompasses several controllers with different media options. The original Am79C970 and the Am79C970A are 10Mbps only devices capable of supporting 10baseT (TP), 10base2 (coax) and 10base5 (AUI) media. The Am79C978 supports only 1Mbps HomePNA media. The Am79C974 is a dual-function SCSI/ethernet chip which is compatible with original Am79C970. All other devices in the family support 10/100Mbps via an internal or external MII PHY.</p>
BOARD LAYOUT	The Am97c97x PCI devices are available in both standalone PCI card format and integrated directly onto the system main board. All configurations are jumperless.
EXTERNAL INTERFACE	<p>The driver provides a vxBus external interface. The only exported routine is the InPciRegister() function, which registers the driver with VxBus.</p>
INCLUDE FILES	am79c97xVxbEnd.h end.h endLib.h netBufLib.h muxLib.h
SEE ALSO	vxBus, ifLib , <i>AMD PCnet/PCI programming manuals</i> , http://www.amd.com

an983VxbEnd

NAME	an983VxbEnd – Infineon AN983B/BX VxBus END driver
ROUTINES	anRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver for the Infineon AN983B/BX PCI 10/100 ethernet controller. The AN983B/BX is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The controller has an embedded 10/100 PHY, with MII management interface.</p> <p>The AN983B controller is designed to be programmed much like the DEC 2114x "tulip" family. It uses the same descriptor layout and RX and TX DMA handling scheme. It differs from the tulip design in two major ways:</p> <ul style="list-style-type: none"> - The AN983B supports MII-based transceivers only (the 2114x supports MII and serial PHYs)

- The AN983B uses a simplified RX filter scheme. The 2114x allowed for several complex filtering schemes using the TX DMA channel to load the multicast hash filter and/or CAM filter table, and it has a 512 bit hash filter. The AN983B's RX filter is programmed entirely through registers, and it has only a 64 bit hash table.

Like the tulip, the AN983B supports both a linked list descriptor mode and a contiguous block mode. In the latter mode, the **next** pointer field can be used as a second data buffer pointer, which can reduce overhead by allowing two packet fragments to be transferred using a single descriptor. This driver uses the contiguous block mode both for the reduction in transfer overhead, and for code simplicity.

The AN983B is often available on consumer NICs, such as the Linksys LNE100TX v4.x.

Note that like the tulip on which it's based, the AN983B can only perform RX DMA to buffers that are 32-bit aligned. Because the ethernet frame header is only 14 bytes in size, this causes the payload to be misaligned, which can lead to unaligned accesses within the VxWorks TCP/IP stack (which uses 32-bit loads and stores to access the address fields in the IP header). On the x86, PPC and Coldfire architectures, these misaligned accesses can be safely ignored, but on all other architectures, the driver is forced to copy received buffers to fix up the alignment before passing them to the stack.

BOARD LAYOUT

EXTERNAL INTERFACE

INCLUDE FILES none

SEE ALSO vxBus, **ifLib**, "Infineon AN983B/BX Datasheet,
http://www.infineon.com/upload/Document/AN983B_X_DS_green_version_PQFP_1.pdf
 f"

bLib

NAME bLib – buffer manipulation library

ROUTINES **bcmp()** – compare one buffer to another
binvert() – invert the order of bytes in a buffer
bswap() – swap buffers
swab() – swap bytes
uswab() – swap bytes with buffers that are not necessarily aligned
bzero() – zero out a buffer
bcopy() – copy one buffer to another

bcopyBytes() – copy one buffer to another one byte at a time
bcopyWords() – copy one buffer to another one word at a time
bcopyLongs() – copy one buffer to another one long word at a time
bfill() – fill a buffer with a specified character
bfillBytes() – fill buffer with a specified character one byte at a time
index() – find the first occurrence of a character in a string
rindex() – find the last occurrence of a character in a string

DESCRIPTION	<p>This library contains routines to manipulate buffers of variable-length byte arrays. Operations are performed on long words when possible, even though the buffer lengths are specified in bytes. This occurs only when source and destination buffers start on addresses that are both odd or both even. If one buffer is even and the other is odd, operations must be done one byte at a time, thereby slowing down the process.</p> <p>Certain applications, such as byte-wide memory-mapped peripherals, may require that only byte operations be performed. For this purpose, the routines bcopyBytes() and bfillBytes() provide the same functions as bcopy() and bfill(), but use only byte-at-a-time operations. These routines do not check for null termination.</p>
INCLUDE FILES	string.h
SEE ALSO	ansiString

bcm52xxPhy

NAME	bcm52xxPhy – driver for Broadcom bcm52xx 10/100 ethernet PHY chips
ROUTINES	bmtPhyRegister() – register with the VxBus subsystem
DESCRIPTION	This file implements the vxBus driver for Broadcom bcm52xx ethernet PHY device. It provides the initialization and functionality routines for this device.
INCLUDE FILES	none

bootInit

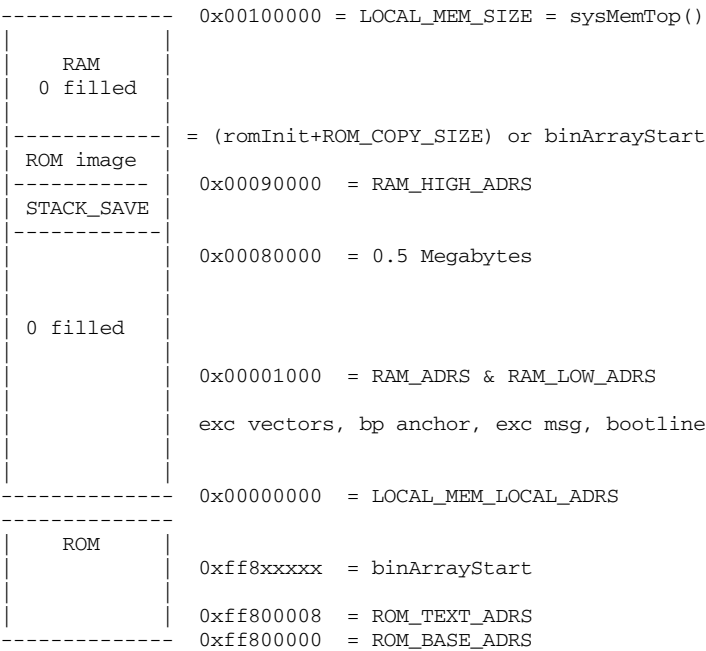
NAME	bootInit – ROM initialization module
ROUTINES	romStart() – generic ROM initialization

DESCRIPTION This module provides a generic boot ROM facility. The target-specific **romInit.s** module performs the minimal preliminary board initialization and then jumps to the C routine **romStart()**. This routine, still executing out of ROM, copies the first stage of the startup code to a RAM address and jumps to it. The next stage clears memory and then uncompresses the remainder of ROM into the final VxWorks ROM image in RAM.

A modified version of the Public Domain **zlib** library is used to uncompress the VxWorks boot ROM executable linked with it. Compressing object code typically achieves over 55% compression, permitting much larger systems to be burned into ROM. The only expense is the added few seconds delay while the first two stages complete.

ROM AND RAM MEMORY LAYOUT

Example memory layout for a 1-megabyte board:



AUTHOR The original compression software for zlib was written by Jean-loup Gailly and Mark Adler. See the manual pages of inflate and deflate for more information on their freely available compression software.

INCLUDE FILES none

SEE ALSO **inflate()**, **romInit()**, and **deflate**

bootLib

NAME	bootLib – boot ROM subroutine library
ROUTINES	bootStructToString() – construct a boot line bootParamsShow() – display boot line parameters bootParamsPrompt() – prompt for boot line parameters
DESCRIPTION	<p>This library contains routines for manipulating a boot line. Routines are provided to construct, print, and prompt for a boot line.</p> <p>When VxWorks is first booted, certain parameters can be specified, such as network addresses, boot device, host, and start-up file. This information is encoded into a single ASCII string known as the boot line. The boot line is placed at a known address (specified in config.h) by the boot ROMs so that the system being booted can discover the parameters that were used to boot the system. The boot line is the only means of communication from the boot ROMs to the booted system.</p> <p>The boot line is of the form:</p> <pre>bootdev(unitnum,procnum)hostname:filename e=# b=# h=# g=# u=userid pw=passwd f=# tn=targetname s=startupscript#rtp.vxe o=other</pre> <p>where:</p> <p><i>bootdev</i> the boot device (required); for example, "ex" for Excelan Ethernet, "bp" for backplane. For the backplane, this field can have an optional anchor address specification of the form "bp=<i>adrs</i>" (see bootBpAnchorExtract()).</p> <p><i>unitnum</i> the unit number of the boot device (0..n).</p> <p><i>procnum</i> the processor number on the backplane, 0..n (required for VME boards).</p> <p><i>hostname</i> the name of the boot host (required).</p> <p><i>filename</i> the file to be booted (required).</p> <p><i>e</i> the Internet address of the Ethernet interface. This field can have an optional subnet mask of the form <i>inet_adrs:subnet_mask</i>. If DHCP is used to obtain the configuration parameters, lease timing information may also be present. This information takes the form <i>lease_duration:lease_origin</i> and is appended to the end of the field. (see bootNetmaskExtract() and bootLeaseExtract()).</p>

- b**
the Internet address of the backplane interface. This field can have an optional subnet mask and/or lease timing information as "e".
- h**
the Internet address of the boot host.
- g**
the Internet address of the gateway to the boot host. Leave this parameter blank if the host is on same network.
- u**
a valid user name on the boot host.
- pw**
the password for the user on the host. This parameter is usually left blank. If specified, FTP is used for file transfers.
- f**
the system-dependent configuration flags. This parameter contains an **or** of option bits defined in **sysLib.h**.
- tn**
the name of the system being booted
- s**
the name of a file to be executed as a start-up script. In addition, if a # separator is used, this parameter can contain a list of RTPs to start.
- o**
"other" string for use by the application.

The Internet addresses are specified in "dot" notation (e.g., 90.0.0.2). The order of assigned values is arbitrary.

EXAMPLE

```
enp(0,0)host:/usr/wpwr/target/config/mz7122/vxWorks e=90.0.0.2 b=91.0.0.2
h=100.0.0.4 g=90.0.0.3 u=bob pw=realtime f=2 tn=target
s=host:/usr/bob/startup#/romfs/helloworld.vxe o=any_string
```

INCLUDE FILES **bootLib.h**

SEE ALSO **bootConfig, bootParseLib.c, usrRtpAppInitBootline.c**

bootParseLib

NAME **bootParseLib** – boot ROM bootline interpreter library

ROUTINES	bootStringToStructAdd() – interpret the boot parameters from the boot line bootStringToStruct() – interpret the boot parameters from the boot line bootLeaseExtract() – extract the lease information from an Internet address bootNetmaskExtract() – extract the net mask field from an Internet address bootBpAnchorExtract() – extract a backplane address from a device field
DESCRIPTION	This library contains routines for interpreting a boot line.
INCLUDE FILES	bootLib.h
SEE ALSO	bootLib.c

cacheArchLib

NAME	cacheArchLib – architecture-specific cache management library
ROUTINES	cacheArchLibInit() – initialize the cache library cacheArchClearEntry() – clear an entry from a cache (68K, x86) cacheStoreBufEnable() – enable the store buffer (MC68060 only) cacheStoreBufDisable() – disable the store buffer (MC68060 only)
DESCRIPTION	<p>This library contains architecture-specific cache library functions for the following processor cache families: Motorola 68K, Intel x86, PowerPC, ARM, and the Solaris and Windows simulators. Each routine description indicates which architecture families support it. Within families, different members support different cache mechanisms; thus, some operations cannot be performed by certain processors because they lack particular functionalities. In such cases, the routines in this library return ERROR. Processor-specific constraints are addressed in the manual entries for routines in this library. If the caches are unavailable or uncontrollable, the routines return ERROR. The exception to this rule is the 68020; although the 68020 has no cache, data cache operations return OK.</p> <p>The MIPS architecture family has cache-related routines in individual BSP libraries. See the reference pages for the individual libraries and routines.</p>
INCLUDE FILES	cacheLib.h, mmuLib.h (ARM only)
SEE ALSO	cacheLib, vmLib

cacheAuLib

NAME	cacheAuLib – Alchemy Au cache management library
ROUTINES	cacheAuLibInit() – initialize the Au cache library
DESCRIPTION	<p>This library contains architecture-specific cache library functions for the Alchemy Au architecture. The Au utilizes a variable-size instruction and data cache that operates in write-through mode. Cache line size also varies.</p> <p>For general information about caching, see the manual entry for cacheLib.</p>
INCLUDE FILES	cacheLib.h
SEE ALSO	cacheLib

cacheLib

NAME	cacheLib – cache management library
ROUTINES	<p>cacheLibInit() – initialize the cache library for a processor architecture</p> <p>cacheEnable() – enable the specified cache</p> <p>cacheDisable() – disable the specified cache</p> <p>cacheLock() – lock all or part of a specified cache</p> <p>cacheUnlock() – unlock all or part of a specified cache</p> <p>cacheFlush() – flush all or some of a specified cache</p> <p>cacheInvalidate() – invalidate all or some of a specified cache</p> <p>cacheClear() – clear all or some entries from a cache</p> <p>cachePipeFlush() – flush processor write buffers to memory</p> <p>cacheTextLocalUpdate() – synchronize the caches on local cpu only</p> <p>cacheTextUpdate() – synchronize the instruction and data caches</p> <p>cacheDmaMalloc() – allocate a cache-safe buffer for DMA devices and drivers</p> <p>cacheDmaFree() – free the buffer acquired with cacheDmaMalloc()</p> <p>cacheDrvFlush() – flush the data cache for drivers</p> <p>cacheDrvInvalidate() – invalidate data cache for drivers</p> <p>cacheDrvVirtToPhys() – translate a virtual address for drivers</p> <p>cacheDrvPhysToVirt() – translate a physical address for drivers</p> <p>cacheForeignFlush() – flush foreign data from selected cache</p> <p>cacheForeignClear() – clear foreign data from selected cache</p> <p>cacheForeignInvalidate() – invalidate foreign data from selected cache</p>

DESCRIPTION

This library provides architecture-independent routines for managing the instruction and data caches. Architecture-dependent routines are documented in the architecture-specific libraries.

The cache library is initialized by **cacheLibInit()** in **usrInit()**. The **cacheLibInit()** routine typically calls an architecture-specific initialization routine in one of the architecture-specific libraries. The initialization routine places the cache in a known and quiescent state, ready for use, but not yet enabled. Cache devices are enabled and disabled by calls to **cacheEnable()** and **cacheDisable()**, respectively.

The structure **CACHE_LIB** in **cacheLib.h** provides a function pointer that allows for the installation of different cache implementations in an architecture-independent manner. If the processor family allows more than one cache implementation, the board support package (BSP) must select the appropriate cache library using the function pointer **sysCacheLibInit**. The **cacheLibInit()** routine calls the initialization function attached to **sysCacheLibInit** to perform the actual **CACHE_LIB** function pointer initialization (see **cacheLib.h**). Note that **sysCacheLibInit** must be initialized when declared; it need not exist for architectures with a single cache design. Systems without caches have all **NULL** pointers in the **CACHE_LIB** structure. For systems with bus snooping, NULLifying the flush and invalidate function pointers in **sysHwInit()** improves overall system and driver performance.

Function pointers also provide a way to supplement the cache library or attach user-defined cache functions for managing secondary cache systems.

Parameters specified by **cacheLibInit()** are used to select the cache mode, either write-through (**CACHE_WRITETHROUGH**) or copyback (**CACHE_COPYBACK**), as well as to implement all other cache configuration features via software bit-flags. Note that combinations, such as setting copyback and write-through at the same time, do not make sense.

Typically, the first argument passed to cache routines after initialization is the **CACHE_TYPE**, which selects the data cache (**DATA_CACHE**) or the instruction cache (**INSTRUCTION_CACHE**).

Several routines accept two additional arguments: an address and the number of bytes. Some cache operations can be applied to the entire cache (bytes = **ENTIRE_CACHE**) or to a portion of the cache. This range specification allows the cache to be selectively locked, unlocked, flushed, invalidated, and cleared. The two complementary routines, **cacheDmaMalloc()** and **cacheDmaFree()**, are tailored for efficient driver writing. The **cacheDmaMalloc()** routine attempts to return a "cache-safe" buffer, which is created by the MMU and a set of flush and invalidate function pointers. Examples are provided below in the section "Using the Cache Library."

Most routines in this library return a **STATUS** value of **OK**, or **ERROR** if the cache selection is invalid or the cache operation fails.

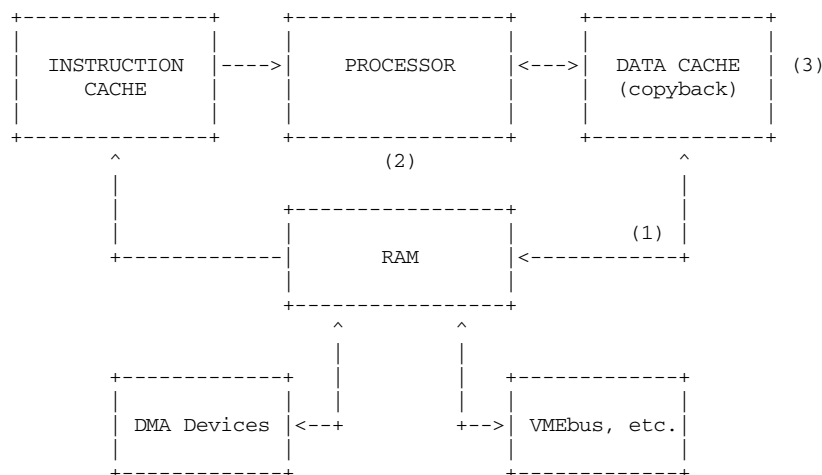
CONFIGURING VXWORKS

To use the cache library, configure VxWorks with the `INCLUDE_CACHE_SUPPORT` and `INCLUDE_CACHE_ENABLE` components.

BACKGROUND

The emergence of RISC processors and effective CISC caches has made cache and MMU support a key enhancement to VxWorks. The VxWorks cache strategy is to maintain coherency between the data cache and RAM and between the instruction and data caches. VxWorks also preserves overall system performance. The product is designed to support several architectures and board designs, to have a high-performance implementation for drivers, and to make routines functional for users, as well as within the entire operating system. The lack of a consistent cache design, even within architectures, has required designing for the case with the greatest number of coherency issues (Harvard architecture, copyback mode, DMA devices, multiple bus masters, and no hardware coherency support).

Caches run in two basic modes, write-through and copyback. The write-through mode forces all writes to the cache and to RAM, providing partial coherency. Writing to RAM every time, however, slows down the processor and uses bus bandwidth. The copyback mode conserves processor performance time and bus bandwidth by writing only to the cache, not RAM. Copyback cache entries are only written to memory on demand. A Least Recently Used (LRU) algorithm is typically used to determine which cache line to displace and flush. Copyback provides higher system performance, but requires more coherency support. Below is a logical diagram of a cached system to aid in the visualization of the coherency issues.



The loss of cache coherency for a VxWorks system occurs in three places:

- (1) data cache / RAM
- (2) instruction cache / data cache
- (3) shared cache lines

A problem between the data cache and RAM (1) results from asynchronous accesses (reads and writes) to the RAM by the processor and other masters. Accesses by DMA devices and alternate bus masters (shared memory) are the primary causes of incoherency, which can be remedied with minor code additions to the drivers.

The instruction cache and data cache (2) can get out of sync when the loader, the debugger, and the interrupt connection routines are being used. The instructions resulting from these operations are loaded into the data cache, but not necessarily the instruction cache, in which case there is a coherency problem. This can be fixed by "flushing" the data cache entries to RAM, then "invalidating" the instruction cache entries. The invalid instruction cache tags will force the retrieval of the new instructions that the data cache has just flushed to RAM.

Cache lines that are shared (3) by more than one task create coherency problems. These are manifest when one thread of execution invalidates a cache line in which entries may belong to another thread. This can be avoided by allocating memory on a cache line boundary, then rounding up to a multiple of the cache line size.

The best way to preserve cache coherency with optimal performance (Harvard architecture, copyback mode, no software intervention) is to use hardware with bus snooping capabilities. The caches, the RAM, the DMA devices, and all other bus masters are tied to a physical bus where the caches can "snoop" or watch the bus transactions. The address cycle and control (read/write) bits are broadcast on the bus to allow snooping. Data transfer cycles are deferred until absolutely necessary. When one of the entries on the physical side of the cache is modified by an asynchronous action, the cache(s) marks its entry(s) as invalid. If an access is made by the processor (logical side) to the now invalid cached entry, it is forced to retrieve the valid entry from RAM. If while in copyback mode the processor writes to a cached entry, the RAM version becomes stale. If another master attempts to access that stale entry in RAM, the cache with the valid version pre-empts the access and writes the valid data to RAM. The interrupted access then restarts and retrieves the now-valid data in RAM. Note that this configuration allows only one valid entry at any time. At this time, only a few boards provide the snooping capability; therefore, cache support software must be designed to handle incoherency hazards without degrading performance.

The determinism, interrupt latency, and benchmarks for a cached system are exceedingly difficult to specify (best case, worst case, average case) due to cache hits and misses, line flushes and fills, atomic burst cycles, global and local instruction and data cache locking, copyback versus write-through modes, hardware coherency support (or lack of), and MMU operations (table walks, TLB locking).

USING THE CACHE LIBRARY

The coherency problems described above can be overcome by adding cache support to existing software. For code segments that are not time-critical (loader, debugger, interrupt connection), the following sequence should be used first to flush the data cache entries and then to invalidate the corresponding instruction cache entries.

```
cacheFlush (DATA_CACHE, address, bytes);  
cacheInvalidate (INSTRUCTION_CACHE, address, bytes);
```

For time-critical code, implementation is up to the driver writer. The following are tips for using the VxWorks cache library effectively.

Incorporate cache calls in the driver program to maintain overall system performance. The cache may be disabled to facilitate driver development; however, high-performance production systems should operate with the cache enabled. A disabled cache will dramatically reduce system performance for a completed application.

Buffers can be static or dynamic. Mark buffers "non-cacheable" to avoid cache coherency problems. This usually requires MMU support. Dynamic buffers are typically smaller than their static counterparts, and they are allocated and freed often. When allocating either type of buffer, it should be designated non-cacheable; however, dynamic buffers should be marked "cacheable" before being freed. Otherwise, memory becomes fragmented with numerous non-cacheable dynamic buffers.

Alternatively, use the following flush/invalidate scheme to maintain cache coherency.

```
cacheInvalidate (DATA_CACHE, address, bytes); /* input buffer */
cacheFlush (DATA_CACHE, address, bytes);      /* output buffer */
```

The principle is to flush output buffers before each use and invalidate input buffers before each use. Flushing only writes modified entries back to RAM, and instruction cache entries never get modified.

Several flush and invalidate macros are defined in **cacheLib.h**. Since optimized code uses these macros, they provide a mechanism to avoid unnecessary cache calls and accomplish the necessary work (return **OK**). Needless work includes flushing a write-through cache, flushing or invalidating cache entries in a system with bus snooping, and flushing or invalidating cache entries in a system without caches. The macros are set to reflect the state of the cache system hardware and software.

Example 1

The following example is of a simple driver that uses **cacheFlush()** and **cacheInvalidate()** from the cache library to maintain coherency and performance. There are two buffers (lines 3 and 4), one for input and one for output. The output buffer is obtained by the call to **memalign()**, a special version of the well-known **malloc()** routine (line 6). It returns a pointer that is rounded down and up to the alignment parameter's specification. Note that cache lines should not be shared, therefore **_CACHE_ALIGN_SIZE** is used to force alignment. If the memory allocator fails (line 8), the driver will typically return **ERROR** (line 9) and quit.

The driver fills the output buffer with initialization information, device commands, and data (line 11), and is prepared to pass the buffer to the device. Before doing so the driver must flush the data cache (line 13) to ensure that the buffer is in memory, not hidden in the cache. The **drvWrite()** routine lets the device know that the data is ready and where in memory it is located (line 14).

More driver code is executed (line 16), then the driver is ready to receive data that the device has placed in an input buffer in memory (line 18). Before the driver can work with the incoming data, it must invalidate the data cache entries (line 19) that correspond to the input

buffer's data in order to eliminate stale entries. That done, it is safe for the driver to retrieve the input data from memory (line 21). Remember to free (line 23) the buffer acquired from the memory allocator. The driver will return **OK** (line 24) to distinguish a successful from an unsuccessful operation.

```

STATUS drvExample1 ()          /* simple driver - good performance */
{
3: void *      pInBuf;          /* input buffer */
4: void *      pOutBuf;         /* output buffer */

6: pOutBuf = memalign (_CACHE_ALIGN_SIZE, BUF_SIZE);

8: if (pOutBuf == NULL)
9:     return (ERROR);          /* memory allocator failed */

11: /* other driver initialization and buffer filling */

13: cacheFlush (DATA_CACHE, pOutBuf, BUF_SIZE);
14: drvWrite (pOutBuf);          /* output data to device */

16: /* more driver code */

18: cacheClear (DATA_CACHE, pInBuf, BUF_SIZE);
19: pInBuf = drvRead (); /* wait for device data */

21: /* handle input data from device */

23: free (pOutBuf);              /* return buffer to memory pool */
24: return (OK);
}

```

Extending this flush/invalidate concept further, individual buffers can be treated this way, not just the entire cache system. The idea is to avoid unnecessary flush and/or invalidate operations on a per-buffer basis by allocating cache-safe buffers. Calls to **cacheDmaMalloc()** optimize the flush and invalidate function pointers to **NULL**, if possible, while maintaining data integrity.

Example 2

The following example is of a high-performance driver that takes advantage of the cache library to maintain coherency. It uses **cacheDmaMalloc()** and the macros **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE**. A buffer pointer is passed as a parameter (line 2). If the pointer is not **NULL** (line 7), it is assumed that the buffer will not experience any cache coherency problems. If the driver was not provided with a cache-safe buffer, it will get one (line 11) from **cacheDmaMalloc()**. A **CACHE_FUNCS** structure (see **cacheLib.h**) is used to create a buffer that will not suffer from cache coherency problems. If the memory allocator fails (line 13), the driver will typically return **ERROR** (line 14) and quit.

The driver fills the output buffer with initialization information, device commands, and data (line 17), and is prepared to pass the buffer to the device. Before doing so, the driver must flush the data cache (line 19) to ensure that the buffer is in memory, not hidden in the

cache. The routine **drvWrite()** lets the device know that the data is ready and where in memory it is located (line 20).

More driver code is executed (line 22), and the driver is then ready to initiate a device operation to read data into the buffer in memory (line 25). Before doing so, it must invalidate the data cache entries (line 25) that correspond to the input buffer's data in order to eliminate stale entries. That done, it is safe for the driver to handle the input data (line 31), which the driver retrieves from memory. Remember to free the buffer (line 33) acquired from the memory allocator. The driver will return **OK** (line 34) to distinguish a successful from an unsuccessful operation.

```
STATUS drvExample2 (pBuf)          /* simple driver - great performance */
2: void *          pBuf;          /* buffer pointer parameter */

{
5:  if (pBuf != NULL)
{
7:      /* no cache coherency problems with buffer passed to driver */
}
    else
    {
11:        pBuf = cacheDmaMalloc (BUF_SIZE);

13:        if (pBuf == NULL)
14:            return (ERROR);      /* memory allocator failed */
    }

17: /* other driver initialization and buffer filling */

19: CACHE_DMA_FLUSH (pBuf, BUF_SIZE);
20: drvWrite (pBuf);              /* output data to device */

22: /* more driver code */

24: CACHE_DMA_INVALIDATE (pBuf, BUF_SIZE);
25: drvStartRead ();             /* start input operation */

27: /* more driver code */

29: drvWait ();                  /* wait for device data */

31: /* handle input data from device */

33: cacheDmaFree (pBuf); /* return buffer to memory pool */
34: return (OK);
}
```

Do not use **CACHE_DMA_FLUSH** or **CACHE_DMA_INVALIDATE** without first calling **cacheDmaMalloc()**, otherwise the function pointers may not be initialized correctly. Note that this driver scheme assumes all cache coherency modes have been set before driver initialization, and that the modes do not change after driver initialization. The **cacheFlush()** and **cacheInvalidate()** functions can be used at any time throughout the system since they are affiliated with the hardware, not the malloc/free buffer.

A call to **cacheLibInit()** in write-through mode makes the flush function pointers NULL. Setting the caches in copyback mode (if supported) should set the pointer to and call an architecture-specific flush routine. The invalidate and flush macros may be NULLified if the hardware provides bus snooping and there are no cache coherency problems.

Example 3

The next example shows a more complex driver that requires address translations to assist in the cache coherency scheme. The previous example had **a priori** knowledge of the system memory map and/or the device interaction with the memory system. This next driver demonstrates a case in which the virtual address returned by **cacheDmaMalloc()** might differ from the physical address seen by the device. It uses the **CACHE_DMA_VIRT_TO_PHYS** and **CACHE_DMA_PHYS_TO_VIRT** macros in addition to the **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE** macros.

The **cacheDmaMalloc()** routine initializes the buffer pointer (line 3). If the memory allocator fails (line 5), the driver will typically return **ERROR** (line 6) and quit. The driver fills the output buffer with initialization information, device commands, and data (line 8), and is prepared to pass the buffer to the device. Before doing so, the driver must flush the data cache (line 10) to ensure that the buffer is in memory, not hidden in the cache. The flush is based on the virtual address since the processor filled in the buffer. The **drvWrite()** routine lets the device know that the data is ready and where in memory it is located (line 11). Note that the **CACHE_DMA_VIRT_TO_PHYS** macro converts the buffer's virtual address to the corresponding physical address for the device.

More driver code is executed (line 13), and the driver is then ready to initiate a device operation to read data into a buffer in memory (line 18). Before doing so, it must invalidate the data cache entries (line 17) that correspond to the input buffer's data in order to eliminate stale entries. Note the use of the **CACHE_DMA_PHYS_TO_VIRT** macro (line 16) on the buffer pointer received from the device. That done, it is safe for the driver to perform the operation (line 18) and handle the input data (line 20), which it retrieves from memory. Remember to free (line 22) the buffer acquired from the memory allocator. The driver will return **OK** (line 23) to distinguish a successful from an unsuccessful operation.

```
STATUS drvExample3 ()           /* complex driver - great performance */ {
2: void * pBufP;
3: void * pBufV = cacheDmaMalloc (BUF_SIZE);

5: if (pBufV == NULL)
6:     return (ERROR);          /* memory allocator failed */

8: /* other driver initialization and buffer filling */

10: CACHE_DMA_FLUSH (pBufV, BUF_SIZE);
11: drvWrite (CACHE_DMA_VIRT_TO_PHYS (pBufV));

13: /* more driver code */

15: pBufP = drvLocateInputBuffer ();
16: pBufV = CACHE_DMA_PHYS_TO_VIRT (pBufP);
17: CACHE_DMA_INVALIDATE (pBufV, BUF_SIZE);
```

```
18: drvRead (pBufP);

20: /* handle input data from device */

22: cacheDmaFree (pBufV);          /* return buffer to memory pool */
23: return (OK);
}
```

Driver Summary

The virtual-to-physical and physical-to-virtual function pointers associated with **cacheDmaMalloc()** are supplements to a cache-safe buffer. Since the processor operates on virtual addresses and the devices access physical addresses, discrepant addresses can occur and might prevent DMA-type devices from being able to access the allocated buffer. Typically, the MMU is used to return a buffer that has pages marked as non-cacheable. An MMU is used to translate virtual addresses into physical addresses, but it is not guaranteed that this will be a "transparent" translation.

When **cacheDmaMalloc()** does something that makes the virtual address different from the physical address needed by the device, it provides the translation procedures. This is often the case when using translation lookaside buffers (TLB) or a segmented address space to inhibit caching (e.g., by creating a different virtual address for the same physical space.) If the virtual address returned by **cacheDmaMalloc()** is the same as the physical address, the function pointers are made NULL so that no calls are made when the macros are expanded.

Board Support Packages

Each board for an architecture with more than one cache implementation has the potential for a different cache system. Hence the BSP for selecting the appropriate cache library. The function pointer **sysCacheLibInit** is set to **cacheXxxLibInit()** ("Xxx" refers to the chip-specific name of a library or function) so that the function pointers for that cache system will be initialized and the linker will pull in only the desired cache library. Below is an example of **cacheXxxLib** being linked in by **sysLib.c**. For systems without caches and for those architectures with only one cache design, there is no need for the **sysCacheLibInit** variable.

```
FUNCPTR sysCacheLibInit = (FUNCPTR) cacheXxxLibInit;
```

For cache systems with bus snooping, the flush and invalidate macros should be NULLified to enhance system and driver performance in **sysHwInit()**.

```
void sysHwInit ()
{
    ...
    cacheLib.flushRtn = NULL;          /* no flush necessary */
    cacheLib.invalidateRtn = NULL;     /* no invalidate necessary */
    ...
}
```

There may be some drivers that require numerous cache calls, so many that they interfere with the code clarity. Additional checking can be done at the initialization stage to determine if **cacheDmaMalloc()** returned a buffer in non-cacheable space. Remember that

it will return a cache-safe buffer by virtue of the function pointers. Ideally, these are `NULL`, since the MMU was used to mark the pages as non-cacheable. The macros `CACHE_XXX_IS_WRITE_COHERENT` and `CACHE_XXX_IS_READ_COHERENT` can be used to check the flush and invalidate function pointers, respectively.

Write buffers are used to allow the processor to continue execution while the bus interface unit moves the data to the external device. In theory, the write buffer should be smart enough to flush itself when there is a write to non-cacheable space or a read of an item that is in the buffer. In those cases where the hardware does not support this, the software must flush the buffer manually. This often is accomplished by a read to non-cacheable space or a NOP instruction that serializes the chip's pipelines and buffers. This is not really a caching issue; however, the cache library provides a `CACHE_PIPE_FLUSH` macro. External write buffers may still need to be handled in a board-specific manner.

INCLUDE FILES	<code>cacheLib.h</code>
SEE ALSO	Architecture-specific cache-management libraries (<code>cacheXxxLib</code>), and the VxWorks programmer guides.

cacheR10kLib

NAME	<code>cacheR10kLib</code> – MIPS R10000 cache management library
ROUTINES	<code>cacheR10kLibInit()</code> – initialize the R10000 cache library
DESCRIPTION	<p>This library contains architecture-specific cache library functions for the MIPS R10000 architecture. The R10000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.</p> <p>For general information about caching, see the manual entry for <code>cacheLib</code>.</p>
INCLUDE FILES	<code>cacheLib.h</code>
SEE ALSO	<code>cacheLib</code>

cacheR4kLib

NAME	<code>cacheR4kLib</code> – MIPS R4000 cache management library
ROUTINES	<code>cacheR4kLibInit()</code> – initialize the R4000 cache library

DESCRIPTION	<p>This library contains architecture-specific cache library functions for the MIPS R4000 architecture. The R4000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.</p> <p>For general information about caching, see the manual entry for cacheLib.</p>
INCLUDE FILES	cacheLib.h
SEE ALSO	cacheLib

cacheR5kLib

NAME	cacheR5kLib – MIPS R5000 cache management library
ROUTINES	cacheR5kLibInit() – initialize the R5000 cache library
DESCRIPTION	<p>This library contains architecture-specific cache library functions for the MIPS R5000 architecture. The R5000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.</p> <p>For general information about caching, see the manual entry for cacheLib.</p>
INCLUDE FILES	cacheLib.h
SEE ALSO	cacheLib

cacheR7kLib

NAME	cacheR7kLib – MIPS R7000 cache management library
ROUTINES	cacheR7kLibInit() – initialize the R7000 cache library
DESCRIPTION	<p>This library contains architecture-specific cache library functions for the MIPS R7000 architecture. The R7000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.</p> <p>For general information about caching, see the manual entry for cacheLib.</p>
INCLUDE FILES	cacheLib.h

SEE ALSO **cacheLib**

cacheSh7750Lib

NAME **cacheSh7750Lib** – Renesas SH7750 cache management library**ROUTINES** **cacheSh7750LibInit()** – initialize the SH7750 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Renesas SH7750, SH7750R and SH7770 architectures. There is a 8-Kbyte instruction cache and 16-Kbyte operand cache on SH7750 and SH7750R. The 16-Kbyte operand can be divided into 8-Kbyte cache and 8-Kbyte memory. The enhanced cache 2-Way mode supports a 16-Kbyte instruction cache and 32-Kbyte operand cache on SH7750R. The 32-Kbyte operand can be divided into 16-Kbyte cache and 16-Kbyte memory. There is a 32-Kbyte instruction cache and 32-Kbyte operand cache on SH7770. The cache 2-Way mode supports a 16-Kbyte instruction cache and 16-Kbyte operand cache for reducing power consumption. The operand cache operates in write-through or write-back (copyback) mode. Cache line size is fixed at 32 bytes, and the cache address array holds physical addresses as cache tags. Cache entries may be "flushed" by accesses to the address array in privileged mode. There is a write-back buffer which can hold one line of cache entry, and the completion of write-back cycle is assured by accessing to any cache through region on SH7750 and by issuing synco instruction on SH7770.

For general information about caching, see the manual entry for **cacheLib**.

INCLUDE FILES **cacheLib.h****SEE ALSO** **cacheLib**

cacheTx49Lib

NAME **cacheTx49Lib** – Toshiba Tx49 cache management library**ROUTINES** **cacheTx49LibInit()** – initialize the Tx49 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Toshiba Tx49 architecture. The Tx49 utilizes a variable-size instruction and data cache that operates in write-back mode. The cache is four-way set associative and the library allows the cache line size to vary.

For general information about caching, see the manual entry for **cacheLib**.

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cbioLib

NAME **cbioLib** – Cached Block I/O library

ROUTINES **cbioLibInit()** – Initialize CBIO Library
cbioBlkRW() – transfer blocks to or from memory
cbioBytesRW() – transfer bytes to or from memory
cbioBlkCopy() – block to block (sector to sector) transfer routine
cbioIoctl() – perform ioctl operation on device
cbioModeGet() – return the mode setting for CBIO device
cbioModeSet() – set mode for CBIO device
cbioRdyChgdGet() – determine ready status of CBIO device
cbioRdyChgdSet() – force a change in ready status of CBIO device
cbioLock() – obtain CBIO device semaphore.
cbioUnlock() – release CBIO device semaphore.
cbioParamsGet() – fill in **CBIO_PARAMS** structure with CBIO device parameters
cbioShow() – print information about a CBIO device
cbioDevVerify() – verify **CBIO_DEV_ID**
cbioWrapBlkDev() – create CBIO wrapper atop a **BLK_DEV** device
cbioDevCreate() – Initialize a CBIO device (Generic)

DESCRIPTION This library provides the Cached Block Input Output Application Programmers Interface (CBIO API). Libraries such as **dosFsLib**, **rawFsLib**, and **usrFdiskPartLib** use the CBIO API for I/O operations to underlying devices.

This library also provides generic services for CBIO modules. The libraries **dpartCbio**, **dcacheCbio**, and **ramDiskCbio** are examples of CBIO modules that make use of these generic services.

This library also provides a CBIO module that converts **blkIo** driver **BLK_DEV** (**blkIo.h**) interface into CBIO API compliant interface using minimal memory overhead. This lean module is known as the basic **BLK_DEV** to CBIO wrapper module.

CBIO MODULES AND DEVICES

A CBIO module contains code for supporting CBIO devices. The libraries **cbioLib**, **dcacheCbio**, **dpartCbio**, and **ramDiskCbio** are examples of CBIO modules.

A CBIO device is a software layer that provide its master control of I/O to its subordinate. CBIO device layers typically reside logically below a file system and above a storage device. CBIO devices conform to the CBIO API on their master (upper) interface.

CBIO modules provide a CBIO device creation routine used to instantiate a CBIO device. The CBIO modules device creation routine returns a **CBIO_DEV_ID** handle. The **CBIO_DEV_ID** handle is used to uniquely identify the CBIO device layer instance. The user of the CBIO device passes this handle to the CBIO API routines when accessing the device.

The libraries **dosFsLib**, **rawFsLib**, and **usrFdiskPartLib** are considered users of CBIO devices because they use the CBIO API on their subordinate (lower) interface. They do not conform to the CBIO API on their master interface, therefore they are not CBIO modules. They are users of CBIO devices and always reside above CBIO devices in the logical stack.

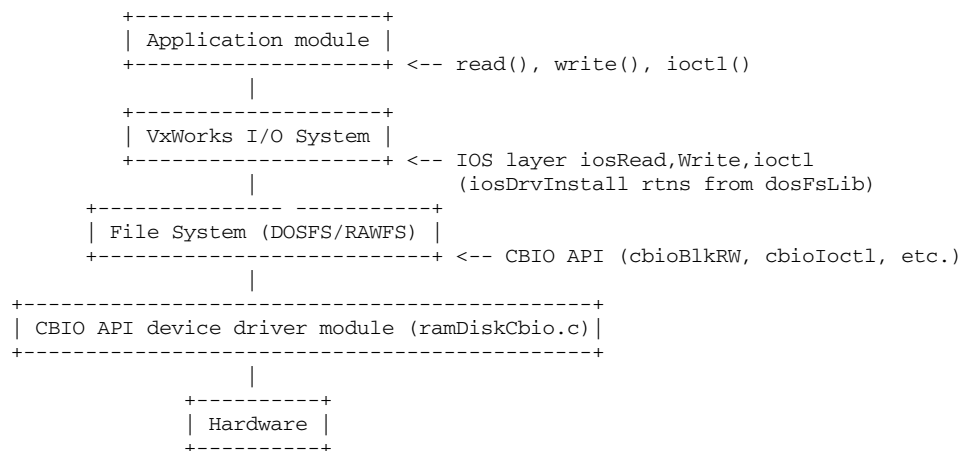
TYPES OF CBIO DEVICES

A "CBIO to CBIO device" uses the CBIO API for both its master and its subordinate interface. Typically, some type of module specific I/O processing occurs during the interface between the master and subordinate layers. The libraries **dpartCbio** and **dcacheCbio** are examples of CBIO to CBIO devices. CBIO to CBIO device layers are stackable. Care should be taken to assemble the stack properly. Refer to each modules reference manual entry for recommendations about the optimum stacking order.

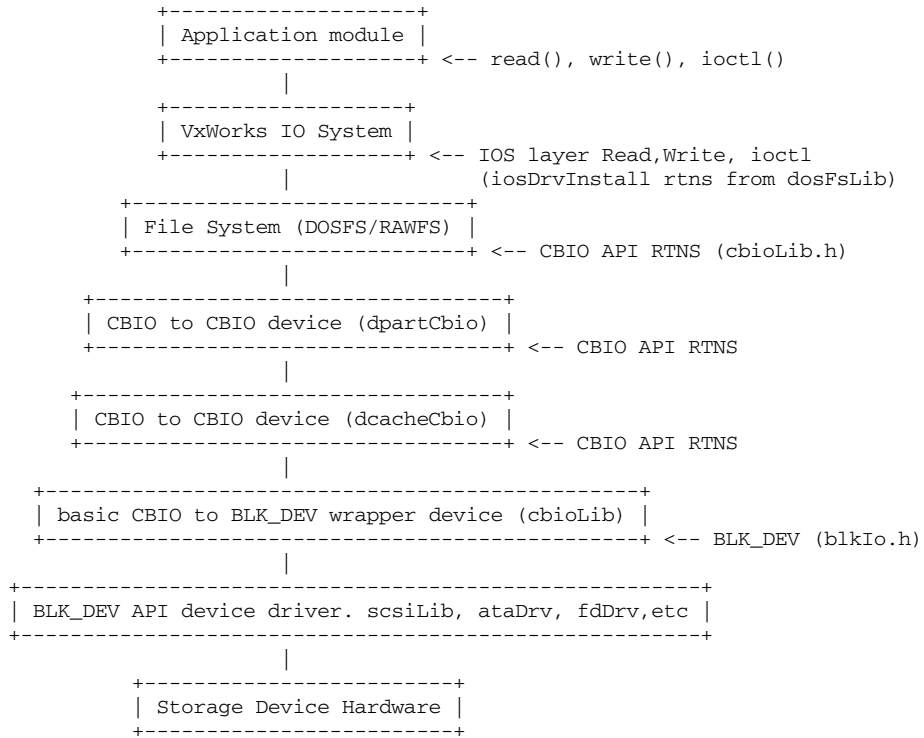
A "CBIO API device driver" is a device driver which provides the CBIO API as the interface between the hardware and its upper layer. The **ramDiskCbio.c** RAM DISK driver is an example of a simple CBIO API device driver.

A "basic **BLK_DEV** to CBIO wrapper device" wraps a subordinate **BLK_DEV** layer with a CBIO API compatible layer. The wrapper is provided via the **cbioWrapBlkDev()** function.

The logical layers of a typical system using a CBIO RAM DISK appear:



The logical layers of a typical system with a fixed disk using CBIO partitioning layer and a CBIO caching layer appears:



PUBLIC CBIO API

The CBIO API provides user access to CBIO devices. Users of CBIO devices are typically either file systems or other CBIO devices.

The CBIO API is exposed via **cbioLib.h**. Users of CBIO modules include the **cbioLib.h** header file. The libraries **dosFsLib**, **dosFsFat**, **dosVDirLib**, **dosDirOldLib**, **usrFdiskPartLib**, and **rawFsLib** all use the CBIO API to access CBIO modules beneath them.

The following functions make up the public CBIO API:

- **cbioLibInit()** - Library initialization routine
- **cbioBlkRW()** - Transfer blocks (sectors) from/to a memory buffer
- **cbioBytesRW()** - Transfer bytes from/to a memory buffer
- **cbioBlkCopy()** - Copy directly from block to block (sector to sector)
- **cbioIoctl()** - Perform I/O control operations on the CBIO device

- **cbioModeGet()** - Get the CBIO device mode (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**)
- **cbioModeSet()** - Set the CBIO device mode (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**)
- **cbioRdyChgdGet()** - Determine the CBIO device ready status state
- **cbioRdyChgdSet()** - Force a change in the CBIO device ready status state
- **cbioLock()** - Obtain exclusive ownership of the CBIO device
- **cbioUnlock()** - Release exclusive ownership of the CBIO device
- **cbioParamsGet()** - Fill a **CBIO_PARAMS** structure with data from the CBIO device
- **cbioDevVerify()** - Verify valid CBIO device
- **cbioWrapBlkDev()** - Create CBIO wrapper atop a **BLK_DEV**
- **cbioShow()** - Display information about a CBIO device

These CBIO API functions (except **cbioLibInit()**) are passed a **CBIO_DEV_ID** handle in the first argument. This handle (obtained from the subordinate CBIO modules device creation routine) is used by the routine to verify the CBIO device is valid and then to perform the requested operation on the specific CBIO device.

When the **CBIO_DEV_ID** passed to the CBIO API routine is not a valid CBIO handle, **ERROR** will be returned with the **errno** set to **S_cbioLib_INVALID_CBIO_DEV_ID** (**cbioLib.h**).

Refer to the individual manual entries for each function for a complete description.

THE BASIC CBIO TO **BLK_DEV** WRAPPER MODULE

The basic CBIO to **BLK_DEV** wrapper is a minimized disk cache using simplified algorithms. It is used to convert a legacy **BLK_DEV** device into as CBIO device. It may be used standalone with solid state disks which do not have mechanical seek and rotational latency delays, such flash cards. It may also be used in conjunction with the **dpartCbio** and **dcacheCbio** libraries. The DOS file system **dosFsDevCreate** routine will call **cbioWrapBlkDev()** internally, so the file system may be installed directly on top of a block driver **BLK_DEV** or it can be used with cache and partitioning support.

The function **cbioWrapBlkDev()** is used to create the CBIO wrapper atop a **BLK_DEV** device.

The functions **dcacheDevCreate** and **dpartDevCreate** also both internally use **cbioDevVerify()** and **cbioWrapBlkDev()** to either stack the new CBIO device atop a validated CBIO device or to create a basic CBIO to **BLK_DEV** wrapper as needed. The user typically never needs to manually invoke the **cbioWrapBlkDev()** or **cbioDevVerify()** functions.

Please note that the basic CBIO **BLK_DEV** wrapper is inappropriate for rotational media without the disk caching layer. The services provided by the **dcacheCbio** module are more appropriate for use on rotational disk devices and will yield superior performance when used.

INCLUDE FILES	cbioLib.h
SEE ALSO	<i>VxWorks Kernel Programmers Guide: I/O System</i>

cdromFsLib

NAME	cdromFsLib – ISO 9660 CD-ROM read-only file system library
ROUTINES	cdromFsInit() – initialize the VxWorks CD-ROM file system cdromFsVolConfigShow() – show the volume configuration information cdromFsVersionDisplay() – display the cdromFs version number cdromFsVersionNumGet() – return the cdromFs version number cdromFsDevDelete() – delete a CD-ROM filesystem (cdromFs) I/O device cdromFsDevCreate() – create a CD-ROM filesystem (cdromFs) I/O device.
DESCRIPTION	<p>This library implements the VxWorks CD-ROM file system (cdromFs). This file system permits the usage of standard POSIX I/O calls, e.g. open(), read(), ioctl(), and close(), to read data from a CD-ROM, CD-R, or CD-RW disk formatted according to the ISO 9660 specification. This library supports multiple devices, concurrent access from multiple tasks, and multiple open files.</p> <p>The component INCLUDE_CDROMFS must be configured into the system to obtain the VxWorks CD-ROM file system (cdromFs).</p> <p>This file system provides access to CD file systems using any standard eXtended Block Device (XBD). Note that the old-style block device (BLK_DEV) is no longer directly supported by cdromFsLib.</p> <p>The creation and deletion of individual instances of cdromFs file systems are typically handled by the file system monitor (FSM). The file system monitor component (INCLUDE_FS_MONITOR) is automatically included in the system when INCLUDE_CDROMFS is included.</p> <p>The underlying XBD driver will raise an insertion event when a CD-ROM device is connected to the target, via a USB port for example, or when media is present in the tray. The file system monitor will handle the insertion event and invoke the appropriate cdromFs create routine to instantiate a cdromFs on the device.</p> <p>Likewise, when the CD-ROM device is disconnected from the target, or when the media is removed, the XBD driver will raise a removal event. This event is handled directly by cdromFsLib, and will result in the deletion of the associated cdromFs.</p> <p>As an example, assume that a target has a CD-ROM drive connected as the master device on the primary ATA controller. In order to instantiate a CD-ROM file system (cdromFs) on this device named <code>"/cdrom"</code>, the INCLUDE_ATA component should be included in the</p>

system, and the `FS_NAMES_ATA_PRIMARY_MASTER` configuration parameter should be set to `"/cdrom"`.

MANUAL INSTANTIATION OF CD-ROM FILE SYSTEMS

The following steps can be followed to "manually" instantiate a CD-ROM file system. For example, this procedure will be required if a custom XBD driver doesn't support the raising of **insertion** and **removal** events as described above.

1. Create an XXX block device (XBD) on the physical device:

```
device_t xxxXbd = xxxXbdDevCreate (...);
```

2. Create a CD-ROM file system (cdromFs) on the XBD device:

```
CDROM_VOL_DESC_ID cdVolDescId = cdromFsDevCreate ("/cdrom", xxxXbd);
```

This will result in the creation of the `"/cdrom"` I/O device, as shown by the output of the **devs** target shell command:

```
-> devs
drv name
 0 /null
 1 /tyCo/0
 2 /aioPipe/0x61723b98
 6 /vio
 7 /tgtsvr
 4 /cdrom
value = 0 = 0x0
```

The CD-ROM file system can be subsequently deleted by the following step.

1. Delete the CD-ROM file system (cdromFs)

```
STATUS status = cdromFsDevDelete (cdVolDescId);
```

In the event that an old-style block device driver (**BLK_DEV**) is being utilized, the "XBD Block Device Wrapper" can be used to translate the **BLK_DEV** interface to an XBD interface. The component **INCLUDE_XBD_BLK_DEV** must be included in a system to obtain the "XBD Block Device Wrapper". The following steps can be followed to "manually" instantiate a CD-ROM using an old-style block device driver.

1. Create an XXX block device (**BLK_DEV**) on the physical device:

```
BLK_DEV * xxxBlkDevId = xxxBlkDevCreate (...);
```

2. Associate an I/O device name with a block device name

```
fsmNameInstall ("xxxDisk0:0", "/cdrom");
```

3. Create an XBD wrapper device around the **BLK_DEV** device:

```
device_t xxxXbd = xbdBlkDevCreate (xxxBlkDevId, "xxxDisk0");
```

The `xbdBlkDevCreate()` invocation will result in the raising of an **insertion** event, i.e. an explicit `cdromFsDevCreate()` is not required.

ISO 9660 FILE AND DIRECTORY NAMING

The strict ISO 9660 specification allows only uppercase file names consisting of 8 characters plus a 3 character suffix.

To accommodate users familiar with MS-DOS, **cdromFsLib** lets you use lowercase name arguments to access files with names consisting entirely of uppercase characters. Mixed-case file and directory names are accessible only if you specify their exact case-correct names.

JOLIET EXTENSIONS FILE AND DIRECTORY NAMING

The Joliet extensions to the ISO 9660 specification are designed to handle long file names up to 340 characters long.

File names must be case correct. The above use of lowercase characters to access files named entirely with uppercase characters is not supported.

The Joliet extensions to **cdromFsLib** do support Unicode file names. Filenames and other identifiers are encoded in 16 bit Unicode (UCS-2) on the media. These are converted to UTF-8 by **cdromFsLib** before being passed back "up" to the I/O system, or before being compared with a string passed "down" from the I/O system.

FILE AND DIRECTORY NAMING COMMON TO ISO 9660 AND THE JOLIET EXTENSIONS

To support multiple versions of the same file, the ISO 9660 specification also supports version numbers. When specifying a file name in an **open()** call, you can select the file version by appending the file name with a semicolon (;) followed by a decimal number indicating the file version. If you omit the version number, **cdromFsLib** opens the latest version of the file.

For the time being, **cdromFsLib** further accommodates MS-DOS users by allowing "\" (backslash) instead of "/" in pathnames. However, the use of the backslash is discouraged because it may not be supported in future versions of **cdromFsLib**.

Finally, **cdromFsLib** uses an 8-bit clean implementation of ISO 9660. Thus, **cdromFsLib** is compatible with CDs using either Latin or Asian characters in the file names.

IOCTL CODES SUPPORTED

FIOGETNAME

Returns the file name for a specific file descriptor.

FIOLABELGET

Retrieves the volume label. This code can be used to verify that a particular volume has been inserted into the drive.

FIOWHERE

Determines the current file position.

FIOWHERE64

Determines the current file position. This is the 64 bit version.

FIOSEEK

Changes the current file position.

FIOSEEK64

Changes the current file position. This is the 64 bit version.

FIONREAD

Tells you the number of bytes between the current location and the end of this file.

FIONREAD64

Tells you the number of bytes between the current location and the end of this file. This is the 64 bit version.

FIOREADDIR

Reads the next directory entry.

FIOUNMOUNT

Announces that the a disk has been removed (all currently open file descriptors are invalidated).

FIOFSTATGET

Gets the file status information (directory entry data).

CDROMFS_DIR_MODE_SET

This is part of the Joliet extensions. It sets the directory mode to the **ioctl()** arg value. That controls whether a file is opened with or without the Joliet extensions. Settings **MODE_ISO9660**, **MODE_JOLIET**, and **MODE_AUTO** do not use Joliet, use Joliet, or try opening the directory first without Joliet and then with Joliet, respectively.

This **ioctl()** unmounts the file system. Thus any open file descriptors are marked obsolete.

CDROMFS_DIR_MODE_GET

This is part of the Joliet extensions. It gets and returns the directory mode set by **CDROMFS_DIR_MODE_SET**.

CDROMFS_STRIP_SEMICOLON

This sets the **readdir()** strip semicolon setting to **FALSE** if arg is 0, and **TRUE** otherwise. If **TRUE**, **readdir()** removes the semicolon and following version number from the directory entries returned.

CDROMFS_GET_VOL_DESC

This returns the primary or supplementary volume descriptor by which the volume is mounted in arg. arg must be type **T_ISO_PVD_SVD_ID** as defined in **cdromFsLib.h**. The result is the volume descriptor adjusted for the endianness of the processor, not the raw volume descriptor from the CD. The result is directly usable by the processor. The result also includes some information not in the volume descriptor, for example which volume descriptor is in-use.

CAVEATS

The VxWorks CD-ROM file system does not support CD sets containing multiple disks.

INCLUDE FILES	cdromFsLib.h
SEE ALSO	ioLib, ISO 9660 Specification, Joliet extension Specification

clockLib

NAME	clockLib – clock library (POSIX)
ROUTINES	clock_getres() – get the clock resolution (POSIX) clock_setres() – set the clock resolution clock_gettime() – get the current time of the clock (POSIX) clock_settime() – set the clock to a specified time (POSIX) clock_nanosleep() – high resolution sleep with specifiable clock
DESCRIPTION	<p>This library provides a clock interface, as defined in the IEEE standard, POSIX 1003.1b.</p> <p>A clock is a software construct that keeps time in seconds and nanoseconds. The clock has a simple interface with three routines: clock_settime(), clock_gettime(), and clock_getres(). The non-POSIX routine clock_setres() that was provided so that clockLib could be informed if there were changes in the system clock rate is no longer necessary. This routine is still present for backward compatibility, but does nothing.</p> <p>Times used in these routines are stored in the timespec structure:</p> <pre>struct timespec { time_t tv_sec; /* seconds */ long tv_nsec; /* nanoseconds (0 -1,000,000,000) */ };</pre> <p>The supported <i>clock_id</i> values are CLOCK_REALTIME, CLOCK_MONOTONIC and CLOCK_THREAD_CPUTIME_ID. Conceivably, additional "virtual" clocks could be supported, or support for additional auxiliary clock hardware (if available) could be added.</p>
CONFIGURATION	To use the POSIX clock library, configure VxWorks with the INCLUDE_POSIX_CLOCKS component.
INCLUDE FILES	timers.h
SEE ALSO	IEEE, the VxWorks programmer guides, and, POSIX 1003.1b documentation.

cnsCompLib

NAME	cnsCompLib – Media type comp library
ROUTINES	cnsCompLibInit() – Initialize the CNS COMP library.
DESCRIPTION	This library is provided to support CNS media type COMP: Connection-Oriented Message Passing protocol which provides a fast method for transferring message across memory boundaries on a single node.
INCLUDE FILES	none

cnsLib

NAME	cnsLib – Component notification system library
ROUTINES	cnsDefaultMediaTypeSet() – Set the default media type. cnsMediumTypeNext() – Return the name of the media type next in the list. cnsAppRegister() – Registers an application with the CNS library. cnsOpen() – Open or create and open named communication medium for read/write. cnsMsgEncode() – Encode a message as understood by CNS. cnsRead() – Read from a communication medium. cnsWrite() – Write to a communication medium. cnsClose() – Close or create and open named communication medium for read/write. cnsMediaRegister() – Registers a communication media with the CNS. cnsLibInit() – Initialize the CNS library. cnsMediaTypeRemove() – Remove a media type from an application's media list.
DESCRIPTION	<p>The component notification system provides a means by which components communicate with each other to exchange events and status information at runtime. Since communicating components could be running in different spaces or in different task contexts or even on different targets, and also on different operating systems, the CNS allows for the configuration of the media type and for distributed messaging services.</p> <p>cnsLib defines two important publicly accessible structures: cnsMediaId_t and cnsMediaInfo_t. Both structures are defined in cnsLib.h and are described below:</p> <pre>typedef struct { char * pName; /* Identifies the media to be used */ long mediaTypeId; /* Media Type - returned by cnsOpen() */ long connId; /* Connection ID - returned by cnsOpen */ } cnsMediaId_t;</pre>

Members of **cnsMediaId_t** are used to identify the media being used during open/close, and read/write accesses.

pName specifies the name of the media together with the address and the name of the component to which the media is bound. The below example illustrates how **pName** can be used:

```
cnsOpen (&mediaId, TRUE);
```

'mediaId can take one of the following forms

```
mediaId.compInfo = "compInfo";           /* Caller assumed to be local.
                                           Use registered local API call */
mediaId.compInfo = "local:/compInfo";     /* Local caller.
                                           Use registered local API call */
mediaId.compInfo = "/compInfo";           /* Caller intra-target.
                                           Use default intra-target media */
mediaId.compInfo = "comp:/compInfo";      /* Use the "comp" media */

mediaId.compInfo = "tipc:<tipcAddress>/compInfo"
                                           /* External caller using the tipc
                                           media. */
mediaId.compInfo = "tcpip:<IP-Address>/compInfo"
                                           /* External caller using the tipc
                                           media. */
```

connId is returned by **cnsOpen()** and it identifies a specific connection via the media.

```
typedef struct
{
    char *                pName;
    CNS_OPEN_FUNCPTR      openFunc;
    CNS_READ_FUNCPTR      readFunc;
    CNS_WRITE_FUNCPTR     writeFunc;
    CNS_CLOSE_FUNCPTR     closeFunc;
    CNS_ACCEPT_FUNCPTR    acceptFunc;
    CNS_IOCTL_FUNCPTR     ioctlFunc;
    CNS_CONN_VALID_FUNCPTR connIsValid;
} cnsMediaInfo_t;
```

Members of the **cnsMediaInfo_t** structure represent the information passed when registering the communication media.

pName identifies the media type.

openFunc is a pointer to the routine to open the media.

readFunc is a pointer to the routine used to read from the media.

writeFunc is a pointer to the routine to write to the media.

closeFunc is a pointer to the routine to close the media.

INCLUDE FILES

cnsLib.h

coreDumpHookLib

NAME	coreDumpHookLib – core dump hook library
ROUTINES	coreDumpCreateHookAdd() – add a routine to be called at every core dump create coreDumpCreateHookDelete() – delete a previously added core dump create routine
DESCRIPTION	<p>This library provides routines for adding extensions to the VxWorks core dump facility. To allow core dump-related facilities to be added to the system without modifying the core dump library, the core dump library provides call-outs every time a core dump is created. The call-outs allow additional routines, or "hooks" to be invoked whenever this event occurs. Those routines can be used to add additional core dump memory filters or to dump additional areas of memory within core dump. Note that those hooks will be called before VxWorks memory is stored within core dump. If one of the core dump creation hooks return an error, the core dump generation is aborted.</p> <p>The hook management routines below allow hooks to be dynamically added to and deleted from the current lists of create hooks:</p> <p>coreDumpCreateHookAdd() Add a routine to be called when a core dump is created.</p> <p>coreDumpCreateHookDelete() Delete a routine from core dump create hook list that was previously added with coreDumpCreateHookAdd()</p>
CONFIGURATION	The core dump hook library is configured in VxWorks whenever core dump facility is included. The maximum number of hooks that can be added is configured by the CORE_DUMP_MAX_HOOKS parameter.
INCLUDE FILES	coreDumpLib.h
SEE ALSO	coreDumpLib , coreDumMemFilterLib , coreDumpShow , the VxWorks Kernel programmer's guide.

coreDumpLib

NAME	coreDumpLib – core dump library
ROUTINES	coreDumpUsrGenerate() – generate a user (on-demand) core dump coreDumpMemDump() – dump an area of memory in VxWorks core dump coreDumpDevFormat() – format the core dump device

DESCRIPTION This library provides the interfaces to the VxWorks Core Dump feature.

The VxWorks Core Dump is an optional feature of the VxWorks kernel that provides the ability to generate a core dump than can be analyzed later by host tools. VxWorks kernel core dumps can be generated in the following situations:

- Fatal System Exception: When an exception occurs during kernel initialization, at interrupt level, or while in VxWorks scheduler. This kind of exception usually causes a system reboot.
- Kernel Panic situations: Unrecoverable situation that is detected by the kernel itself. This kind of error usually causes a system reboot.
- Kernel Task Level Exception: When an exception occurs in a kernel task, but is not fatal to the system (not leading to a target reboot).
- User core dump: on demand user core dump.

CONFIGURATION To enable VxWorks core dump support, configure VxWorks with the `INCLUDE_CORE_DUMP` component.

CORE DUMP SUPPORT CONFIGURATION

By default, the kernel core dump is configured to generate core dumps to persistent memory. The size of the persistent memory reserved for the core dump storage is defined using `CORE_DUMP_MEM_REGION_SIZE` parameter. By default, the `CORE_DUMP_MEM_REGION_SIZE` parameter is set to use all the remaining persistent memory, but it can be adjusted according to your configuration.

The persistent memory must be increased to add the memory core dump storage area. Note that increasing `PM_RESERVED_MEM` reduces the size of memory available for VxWorks execution, so you must make sure that it remains enough memory for VxWorks and application(s). The `pmShow()` routine can be used to check if the persistent memory configuration is correct:

```
-> pmShow
Arena base address: 0x62000000
Arena raw size: 0x06000000
Arena data size: 0x05ffe000
Arena free space: 0x00000000
Region 0 [edrErrorLog]
    offset: 0x00002000
    size: 0x00080000
    address: 0x62002000
Region 1 [CoreDumpStorage]
    offset: 0x00082000
    size: 0x05f7e000
    address: 0x62082000
value = 0 = 0x0
->
```

CORE DUMP COMPRESSION

The core dump facility provides two core dump compression methods: An RLE based compression which can be added by selecting **INCLUDE_CORE_DUMP_COMPRESS_RLE** component (Default compression method). This algorithm has the advantage of being faster than the other method, but it provides lower compression rate. The other compression method is based on ZLIB, and can be added by selecting **INCLUDE_CORE_DUMP_COMPRESS** component. The Zlib compression level can be specified using **CORE_DUMP_COMPRESSION_LEVEL** parameter.

CORE DUMP AUTOMATIC GENERATION

By default, when core dump support is included, core dumps will be generated for the exceptions that are fatal to the system (Leading to a target reboot). This facility can be disabled by setting **FATAL_SYSTEM_CORE_DUMP_ENABLE** parameter to **FALSE**.

By default, the exceptions that are not fatal to the system do not generate core dumps. This feature can be enabled by setting **KERNEL_APPL_CORE_DUMP_ENABLE** parameter to **TRUE**.

CORE DUMP CHECKSUMMING FACILITY

The core dump checksumming facility can be used to verify that a core dump has not been corrupted on device after its generation. To enable the core dump checksum generation the **CORE_DUMP_CKSUM_ENABLE** parameter must be set to **TRUE**. The core dump checksum can then be verified using **coreDumpInfoGet()** or **coreDumpShow()** routines.

CONFIGURING THE GENERIC RAW DEVICE STORAGE LAYER

It is possible to configure core dump support to store core dump in devices like flash devices or ATA disks. In order to use this kind of device, you must include the **INCLUDE_CORE_DUMP_RAW_DEV** component and provide a driver to access this device. For more information, please refer to the Wind River Diagnostic Programmer's Guide.

CORE DUMP STORAGE DEVICE FORMATING

Before being able to generate core dumps, the core dump storage device must be formatted. This must be done using **coreDumpDevFormat()** routine. If the **INCLUDE_CORE_DUMP_SHOW** component is included, the **coreDumpShow()** routine can then be used to display information on the generated core dumps.

CORE DUMP RETRIEVAL

The core dump can be retrieved using the following set of APIs:

coreDumpIsAvailable()
is a core dump available for retrieval

coreDumpNextGet()
get the next core dump on device

coreDumpInfoGet()
get information on a core dump

coreDumpOpen()
open an existing core dump for retrieval

coreDumpClose()
close a core dump

coreDumpRead()
read from a core file

coreDumpCopy()
copy a core dump to the given path

INCLUDE FILES **coreDumpLib.h**

ERRNOS Routines from this library can return the following core dump specific ernnos:

S_coreDumpLib_CORE_DUMP_COMPRESSION_ERROR
There was an error while compressing core dump image

S_coreDumpLib_CORE_DUMP_DEVICE_READ_ERROR
There was an error reading from core dump device.

S_coreDumpLib_CORE_DUMP_DEVICE_WRITE_ERROR
There was an error writing to core dump device.

S_coreDumpLib_CORE_DUMP_DEVICE_ERASE_ERROR
There was an error erasing the core dump device.

S_coreDumpLib_CORE_DUMP_DEVICE_OPEN_ERROR
There was an error opening a core dump on core dump device.

S_coreDumpLib_CORE_DUMP_DEVICE_CLOSE_ERROR
There was an error closing a core dump on core dump device.

S_coreDumpLib_CORE_DUMP_DEVICE_NOT_INITIALIZED
Core dump storage device is not initialized

S_coreDumpLib_CORE_DUMP_DEVICE_TOO_SMALL
Core dump storage device is full or too small

S_coreDumpLib_CORE_DUMP_GENERATE_ALREADY_RUNNING
Core dump generator is already running.

S_coreDumpLib_CORE_DUMP_GENERATE_NOT_RUNNING
Core dump generator is not running

S_coreDumpLib_CORE_DUMP_INVALID_ARGS
Invalid arguments was provided to core dump interface

S_coreDumpLib_CORE_DUMP_INVALID_DEVICE
Device provided to core dump library is not valid.

S_coreDumpLib_CORE_DUMP_STORAGE_NOT_FORMATED
Core dump storage is not formatted

S_coreDumpLib_CORE_DUMP_TOO_MANY_CORE_DUMP

Too many core dumps are already stored in core dump storage

Note that other errnos, not listed here, may come from libraries internally used by the core dump library.

SEE ALSO

coreDumpShow, coreDumpHookLib, coreDumpMemFilterLib, coreDumpUtilLib, Wind River Diagnostic Programmer's Guide

coreDumpMemFilterLib

NAME

coreDumpMemFilterLib – core dump memory filtering library

ROUTINES

coreDumpMemFilterAdd() – add a memory region filter
coreDumpMemFilterDelete() – delete a memory region filter

DESCRIPTION

This library provides the interface to the VxWorks core dump memory filtering facility.

The VxWorks core dump memory filtering allows a user to avoid dumping an area of memory in the core dump. This is useful to limit the size of a generated core dump by removing useless information.

A VxWorks core dump memory filter can be added either at VxWorks startup time or during core dump generation using VxWorks core dump creation hooks.

CONFIGURATION

The VxWorks core dump memory filtering facility is included whenever VxWorks core dump support is included.

The number of core dump memory filter than can be added is limited by the value of **CORE_DUMP_MEM_FILTER_MAX** that can be set at build time. The default value for this parameter is 10.

INCLUDE FILES

coreDumpLib.h

ERRNOS

Routines from this library can return the following core dump specific errnos:

S_coreDumpLib_CORE_DUMP_FILTER_TABLE_FULL

Core dump memory filter table is full

S_coreDumpLib_CORE_DUMP_FILTER_NOT_FOUND

Filter not found in core dump filter table

Note that other errnos, not listed here, may come from libraries internally used by the core dump library.

SEE ALSO

coreDumpLib, coreDumpHookLib, VxWorks Kernel Programmer's Guide

coreDumpShow

NAME	coreDumpShow – core dump show routines
ROUTINES	coreDumpShow() – display information on core dumps coreDumpDevShow() – display information on core dump device
DESCRIPTION	This library provides routines to show VxWorks core dump related information such as generated core dumps available on the core dump storage device and information o the core dump storage device itself.
CONFIGURATION	The routines in this library are included if the INCLUDE_CORE_DUMP_SHOW component is configured into VxWorks.
INCLUDE FILES	coreDumpLib.h
SEE ALSO	coreDumpLib , <i>VxWorks Kernel Programmer's Guide</i>

coreDumpUtilLib

NAME	coreDumpUtilLib – core dump utility library
ROUTINES	coreDumpIsAvailable() – is a core dump available for retrieval coreDumpNextGet() – get the next core dump on device coreDumpInfoGet() – get information on a core dump coreDumpOpen() – open an existing core dump for retrieval coreDumpClose() – close a core dump coreDumpRead() – read from a core file coreDumpCopy() – copy a core dump to the given path
DESCRIPTION	This library provides some core dump utilities to manipulate core dumps available on core dump storage device.
INFORMATION ROUTINES	The coreDumpIsAvailable() routine can be used to determine if at least one core dump is available for retrieval. The list of core dumps available on device can be parsed using coreDumpNextGet() , and coreDumpInfoGet() can be used to get information on each core dump.

RETRIEVAL ROUTINES

If a file system visible from Wind River development tools is available on your VxWorks system, then **coreDumpCopy()** routine can be used to copy the specified core dump or all core dumps to a given directory. If this file system is not available, the **coreDumpOpen()**, **coreDumpRead()** and **coreDumpClose()** routines can be used to read core dumps and transfer them using your preferred method.

INCLUDE FILES **coreDumpLib.h**

ERRNOS Routines from this library can return the following core dump specific errnos:

S_coreDumpLib_CORE_DUMP_INVALID_ARGS
Invalid arguments was provided to core dump interface

S_coreDumpLib_CORE_DUMP_INVALID_CORE_DUMP
Accessed core dump is invalid or corrupted

S_coreDumpLib_CORE_DUMP_PATH_TOO_LONG
Core dump copy path is too long.

S_coreDumpLib_CORE_DUMP_STORAGE_NOT_FORMATED
Device provided to core dump library is not valid.

Note that other errnos, not listed here, may come from libraries internally used by the core dump library.

SEE ALSO **coreDumpLib**, **coreDumpShow**, **coreDumpHookLib**, **coreDumpMemFilterLib**, *VxWorks Kernel Programmer's Guide*

cplusLib

NAME **cplusLib** – basic run-time support for C++

ROUTINES **cplusCallNewHandler()** – call the allocation failure handler (C++)
cplusCtors() – call static constructors (C++)
cplusCtorsLink() – call all linked static constructors (C++)
cplusDemanglerSet() – change C++ demangling mode (C++)
cplusDemanglerStyleSet() – change C++ demangling style (C++)
cplusDtors() – call static destructors (C++)
cplusDtorsLink() – call all linked static destructors (C++)
cplusXtorGet() – get the c++ Xtors strategy
cplusLibInit() – initialize the C++ library (C++)
cplusXtorSet() – change C++ static constructor calling strategy (C++)
operator_delete() – default run-time support for memory deallocation (C++)
operator_new() – default run-time support for operator new (C++)

operator_new() – default run-time support for operator new (nothrow) (C++)
operator_new() – run-time support for operator new with placement (C++)
set_new_handler() – set new_handler to user-defined function (C++)
set_terminate() – set terminate to user-defined function (C++)

DESCRIPTION	<p>This library provides run-time support and shell utilities that support the development of VxWorks applications in C++. The run-time support can be broken into three categories:</p> <ul style="list-style-type: none">- Support for C++ new and delete operators.- Support for initialization and cleanup of static objects. <p>Shell utilities are provided for:</p> <ul style="list-style-type: none">- Resolving overloaded C++ function names.- Hiding C++ name mangling, with support for terse or complete name demangling.- Manual or automatic invocation of static constructors and destructors. <p>The usage of cplusplus is more fully described in the <i>"VxWorks Kernel Programmer's Guide: C++ Development."</i></p>
INCLUDE FILES	none
SEE ALSO	<i>"VxWorks Kernel Programmer's Guide: C++ Development"</i>

cpuPwrLightLib

NAME	cpuPwrLightLib – light power manager library (x86, PPC and VxSim)
ROUTINES	cpuPwrMgrEnable() – Set the CPU light power management to ON/OFF cpuPwrMgrIsEnabled() – Get the CPU power management status
DESCRIPTION	<p>This module provides a light CPU power manager for the x86, PPC and VxSim CPU architectures. The windPwrLib library provides similar capability for ARM. The vxPowerModeSet() and vxPowerModeGet() APIs of the vxLib library provide similar functionality for SH.</p>

The light power manager allows a kernel application to control whether or not the CPU is put in a non-executing state when the VxWorks kernel becomes idle. A non-executing state is a state where the CPU stops fetching and executing instructions. It is often referred to as a sleep state. By putting the CPU to sleep when there are no ISRs to process or tasks to dispatch the light power manager effectively reduces the power consumption of the CPU. The behaviour of this power manager is the same as the power management scheme present in previous versions of VxWorks. For this release of VxWorks the light power

manager is only available for Pentium processors. Power management for other processors continues to be provided by libraries that exist in previous versions of VxWorks.

This module contains two callable APIs. **cpuPwrMgrEnable()** is used to enable and disable the power manager. When enabled the power manager puts the CPU in the C1 state when the VxWorks kernel becomes idle. The "C1 state" is a term borrowed from the Advance Configuration and Power Interface (ACPI) specification. It is a non-executing state. Refer to the VxWorks Kernel Programmer's guide for more details on this subject. When the CPU is in a non-executing state, only an interrupt or another type of asynchronous exception can typically wake up the CPU. Details regarding the events that cause a CPU to wake up and the latency associated with this process can be found in the relevant processor user's manual. The other callable API is **cpuPwrMgrIsEnabled()**. This routine allows the caller to determine the state (enabled/disabled) of the lite power manager.

Kernel applications wishing to migrate from the **vxPowerModeSet()** and **vxPowerModeGet()** API to the API provided by this module can do so in the following manner:

- Replace calls to **vxPowerModeSet(VX_POWER_MODE_DISABLE)** with **cpuPwrMgrEnable(FALSE)**.
- Replace calls to **vxPowerModeSet(VX_POWER_MODE_AUTOHALT)** with **cpuPwrMgrEnable(TRUE)**.
- Replace calls to **vxPowerModeGet()** with **cpuPwrMgrIsEnabled()**. Note that the return value for these two routines are not the same.

The light power manager is configured into VxWorks using either of the following methods:

Using Workbench

With the kernel configurator include the **INCLUDE_CPU_PWR_MGMT** component under the **FOLDER_CPU_PWR_MGMT** folder and select the **INCLUDE_CPU_LIGHT_PWR_MGR** from the **SELECT_CPU_PWR_MGR** selection.

Using the vxprj Command Line Tool

Use the **add** command to include the **INCLUDE_CPU_LIGHT_PWR_MGR** component.

This power manager handles the following event:

CPU_PWR_EVENT_IDLE_ENTER

Power manager does not handle the event directly. It simply tells the framework to set the power state of the CPU to **cpuPwrC1State** when the kernel is idle.

The following power management events are not handled by this power manager:

CPU_PWR_EVENT_INT_ENTER
CPU_PWR_EVENT_INT_EXIT
CPU_PWR_EVENT_IDLE_EXIT
CPU_PWR_EVENT_TASK_SWITCH
CPU_PWR_EVENT_CPU_UTIL
CPU_PWR_EVENT_THRES_CROSS

CPU_PWR_EVENT_PRIORITY_CHANGE

INCLUDE FILES **cpuPwrLib.h, cpuPwrMgr.h**

SEE ALSO **windPwrLib, vxLib, VxWorks Programmer's Guide**

cpuPwrUtilLib

NAME **cpuPwrUtilLib** – utilization-based CPU power manager (x86 only)

ROUTINES

DESCRIPTION This library provides a CPU-utilization based CPU power manager for the x86 CPU architecture. This power manager monitors CPU utilization and adjusts the performance state of the silicon to keep the CPU utilization between a low and high threshold. Depending on the hardware, this is achieved by modifying the operating frequency and/or voltage of the processor.

There are no callable APIs provided by this library. To make use of this power manager simply configure it into VxWorks using either of the following methods:

Using Workbench

With the kernel configurator include the **INCLUDE_CPU_PWR_MGMT** component under the **FOLDER_CPU_PWR_MGMT** folder and select the **INCLUDE_CPU_UTIL_PWR_MGR** from the **SELECT_CPU_PWR_MGR** selection.

Using the vxprj Command Line Tool

Use the *add* command to include the **INCLUDE_CPU_UTIL_PWR_MGR** component.

The low and high CPU utilization thresholds are defined using the **CPU_PWR_DOWN_UTIL** and **CPU_PWR_UP_UTIL** configuration parameters of the **INCLUDE_CPU_UTIL_PWR_MGR** component.

PERFORMANCE CONSIDERATIONS

A non-negligible amount of processing is required for this power manager to monitor the CPU-utilization of the system. This may cause performance degradation that is unacceptable for some systems. Users may want to consider using the light power manager for performance critical systems. See reference entry for **cpuPwrLightLib** for more information.

SMP CONSIDERATIONS

This library is not supported on VxWorks SMP

INCLUDE FILES **cpuPwrLib.h**

SEE ALSO **cpuPwrLightLib**, *VxWorks Programmer's Guide*

cpuset

NAME **cpuset** – cpuset_t type manipulation macros

ROUTINES **CPUSET_SET()** – set a CPU in a CPU set
CPUSET_SETALL() – set all CPUs in a CPU set
CPUSET_SETALL_BUT_SELF() – set all CPUs except self in CPU set
CPUSET_CLR() – clear a CPU from a CPU set
CPUSET_ZERO() – clear all CPUs from a CPU set
CPUSET_ISSET() – determine if a CPU is set in a CPU set
CPUSET_ISZERO() – determine if all CPUs are cleared from a CPU set
CPUSET_ATOMICSET() – atomically set a CPU in a CPU set
CPUSET_ATOMICCLR() – atomically clear a CPU from a CPU set
CPUSET_ATOMICCOPY() – atomically copy a CPU set value

DESCRIPTION This module provides a set of macros to manipulate cpuset_t variables. These are opaque variables and must therefore be read and written to using the macros in this module. The cpuset_t type variable is used to identify CPUs in a set of CPUs. It is used in a number of VxWorks SMP APIs.

INCLUDE FILES **cpuset.h**

SEE ALSO **vxCpuLib**

dbgArchLib

NAME **dbgArchLib** – architecture-dependent debugger library

ROUTINES **a0()** – return the contents of register **a0** (also **a1** - **a7**) (MC680x0)
d0() – return the contents of register **d0** (also **d1** - **d7**) (MC680x0)
sr() – return the contents of the status register (SH)
dbgBpTypeBind() – bind a breakpoint handler to a breakpoint type (MIPS R3000, R4000, R4650)
edi() – return the contents of register **edi** (also **esi** - **eax**) (x86)
eflags() – return the contents of the status register (x86)

r0() – return the contents of register **r0** (also **r1** - **r14**) (ARM)
cpsr() – return the contents of the current processor status register (ARM)
psrShow() – display the meaning of a specified PSR value, symbolically (ARM)
r0() – return the contents of general register **r0** (also **r1**-`r15') (SH)
sr() – return the contents of control register **sr** (also **gbr**, **vbr**) (SH)
mach() – return the contents of system register **mach** (also **macl**, **pr**) (SH)
g0() – return the contents of register **g0** (also **g1**-**g7**) (SimSolaris)
o0() – return the contents of register **o0** (also **o1**-**o7**) (SimSolaris)
l0() – return the contents of register **l0** (also **l1**-**l7**) (SimSolaris)
i0() – return the contents of register **i0** (also **i1**-**i7**) (SimSolaris)
npc() – return the contents of the next program counter (SimSolaris)
psr() – return the contents of the processor status register (SimSolaris)
wim() – return the contents of the window invalid mask register (SimSolaris)
y() – return the contents of the y register (SimSolaris)
edi() – return the contents of register **edi** (also **esi** - **eax**) (x86/SimNT)
eflags() – return the contents of the status register (x86/SimNT)

DESCRIPTION

This module provides architecture-specific support functions for **dbgLib**. It also includes user-callable functions for accessing the contents of registers in a task's TCB (task control block). These routines include:

MIPS:

dbgBpTypeBind() – bind a breakpoint handler to a breakpoint type

x86/SimNT:

edi() - **eax()** – named register values
eflags() – status register value

SH:

r0() - **r15()** – general registers (**r0** - **r15**)
sr() – status register (**sr**)
gbr() – global base register (**gbr**)
vbr() – vector base register (**vbr**)
mach() – multiply and accumulate register high (**mach**)
macl() – multiply and accumulate register low (**macl**)
pr() – procedure register (**pr**)

ARM:

r0() - **r14()** – general-purpose registers (**r0** - **r14**)
cpsr() – current processor status reg (**cpsr**)
psrShow() – **psr** value, symbolically

SimSolaris:

g0() - **g7()** – global registers (**g0** - **g7**)
o0() - **o7()** – out registers (**o0** - **o7**, note lower-case "o")
l0() - **l7()** – local registers (**l0** - **l7**, note lower-case "l")
i0() - **i7()** – in registers (**i0** - **i7**)
npc() – next program counter (**npc**)
psr() – processor status register (**psr**)
wim() – window invalid mask (**wim**)

y() - y register

NOTE	The routine pc() , for accessing the program counter, is found in usrLib .
INCLUDE FILES	none
SEE ALSO	dbgLib

dbgLib

NAME	dbgLib – shell debugging facilities
ROUTINES	dbgInit() – initialize the shell debugging package dbgHelp() – display debugging help menu b() – set or display breakpoints e() – set or display eventpoints (WindView) bh() – set a hardware breakpoint bd() – delete a breakpoint bdall() – delete all breakpoints c() – continue from a breakpoint cret() – continue until the current subroutine returns s() – single-step a task so() – single-step, but step over a subroutine l() – disassemble and display a specified number of instructions tt() – display a stack trace of a task
DESCRIPTION	<p>This library contains VxWorks's primary interactive debugging routines, which provide the following facilities:</p> <ul style="list-style-type: none">- task breakpoints- task single-stepping- symbolic disassembly- symbolic task stack tracing

In addition, **dbgLib** provides the facilities necessary for enhanced use of other VxWorks functions, including:

- enhanced shell abort
- exception handling (via **tyLib** and **excLib**)

The facilities of **excLib** are used by **dbgLib** to support breakpoints, single-stepping, and additional exception handling functions.

INITIALIZATION The debugging facilities provided by this module are optional. In the standard VxWorks development configuration as distributed, the debugging package is included. The configuration macro is **INCLUDE_DEBUG**. When defined, it enables the call to **dbgInit()** in the VxWorks initialisation task. The **dbgInit()** routine initializes **dbgLib** and must be made before any other routines in the module are called.

BREAKPOINTS Use the routine **b()** or **bh()** to set breakpoints. Breakpoints can be set to be hit by a specific task or all tasks. Multiple breakpoints for different tasks can be set at the same address. Clear breakpoints with **bd()** and **bdall()**.

When a task hits a breakpoint, the task is suspended and a message is displayed on the console. At this point, the task can be examined, traced, deleted, its variables changed, etc. If you examine the task at this point (using the **i()** routine), you will see that it is in a suspended state. The instruction at the breakpoint address has not yet been executed.

To continue executing the task, use the **c()** routine. The breakpoint remains until it is explicitly removed.

EVENTPOINTS (WINDVIEW)

When WindView is installed, **dbgLib** supports eventpoints. Use the routine **e()** to set eventpoints. Eventpoints can be set to be hit by a specific task or all tasks. Multiple eventpoints for different tasks can be set at the same address.

When a task hits an eventpoint, an event is logged and is displayed by VxWorks kernel instrumentation.

You can manage eventpoints with the same facilities that manage breakpoints: for example, unbreakable tasks (discussed below) ignore eventpoints, and the **b()** command (without arguments) displays eventpoints as well as breakpoints. As with breakpoints, you can clear eventpoints with **bd()** and **bdall()**.

UNBREAKABLE TASKS

An *unbreakable* task ignores all breakpoints. Tasks can be spawned unbreakable by specifying the task option **VX_UNBREAKABLE**. Tasks can subsequently be set unbreakable or breakable by resetting **VX_UNBREAKABLE** with **taskOptionsSet()**. Several VxWorks tasks are spawned unbreakable, such as the shell, the exception support task **excTask()**, and several network-related tasks.

DISASSEMBLER AND STACK TRACER

The **l()** routine provides a symbolic disassembler. The **tt()** routine provides a symbolic stack tracer.

SHELL ABORT AND EXCEPTION HANDLING

This package includes enhanced support for the shell in a debugging environment. The terminal abort function, which restarts the shell, is invoked with the abort key if the **OPT_ABORT** option has been set. By default, the abort key is CTRL-C. For more information, see the manual entries for **tyAbortSet()** and **tyAbortFuncSet()**.

THE DEFAULT TASK AND TASK REFERENCING

Many routines in this module take an optional task name or ID as an argument. If this argument is omitted or zero, the "current" task is used. The current task (or "default" task) is the last task referenced. The **dbgLib** library uses **taskIdDefault()** to set and get the last-referenced task ID, as do many other VxWorks routines.

All VxWorks shell expressions can reference a task by either ID or name. The shell attempts to resolve a task argument to a task ID; if no match is found in the system symbol table, it searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

INCLUDE FILES **dbgLib.h**

SEE ALSO **excLib**, **tyLib**, **taskIdDefault()**, **taskOptionsSet()**, **tyAbortSet()**, **tyAbortFuncSet()**, **windsh**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

dcacheCbio

NAME **dcacheCbio** – Disk Cache Driver

ROUTINES **dcacheDevCreate()** – Create a disk cache
 dcacheDevDisable() – Disable the disk cache for this device
 dcacheDevEnable() – Reenable the disk cache
 dcacheDevTune() – modify tunable disk cache parameters
 dcacheDevMemResize() – set a new size to a disk cache device
 dcacheShow() – print information about disk cache
 dcacheHashTest() – test hash table integrity

DESCRIPTION This module implements a disk cache mechanism via the CBIO API. This is intended for use by the VxWorks DOS file system, to store frequently used disk blocks in memory. The disk cache is unaware of the particular file system format on the disk, and handles the disk as a collection of blocks of a fixed size, typically the sector size of 512 bytes.

The disk cache may be used with SCSI, IDE, ATA, Floppy or any other type of disk controllers. The underlying device driver may be either comply with the CBIO API or with the older block device API.

This library interfaces to device drivers implementing the block device API via the basic CBIO **BLK_DEV** wrapper provided by **cbioLib**.

Because the disk cache complies with the CBIO programming interface on both its upper and lower layers, it is both an optional and a stackable module. It can be used or omitted depending on resources available and performance required.

The disk cache module implements the CBIO API, which is used by the file system module to access the disk blocks, or to access bytes within a particular disk block. This allows the file system to use the disk cache to store file data as well as Directory and File Allocation Table blocks, on a Most Recently Used basis, thus keeping a controllable subset of these disk structures in memory. This results in minimized memory requirements for the file system, while avoiding any significant performance degradation.

The size of the disk cache, and thus the memory consumption of the disk subsystem, is configured at the time of initialization (see **dcacheDevCreate()**), allowing the user to trade-off memory consumption versus performance. Additional performance tuning capabilities are available through **dcacheDevTune()**.

Briefly, here are the main techniques deployed by the disk cache:

- Least Recently Used block re-use policy
- Read-ahead
- Write-behind with sorting and grouping
- Hidden writes
- Disk cache bypass for large requests
- Background disk updating (flushing changes to disk) with an adjustable update period (ioctl flushes occur without delay.)

Some of these techniques are discussed in more detail below; others are described in various professional and academic publications.

DISK CACHE ALGORITHM

The disk cache is composed internally of a number cache blocks, of the same size as the disk physical block (sector). These cache blocks are maintained in a list in "Most Recently Used" order, that is, blocks which are used are moved to the top of this list. When a block needs to be relinquished, and made available to contain a new disk block, the Least Recently Used block will be used for this purpose.

In addition to the regular cache blocks, some of the memory allocated for cache is set aside for a "big buffer", which may range from 1/4 of the overall cache size up to 64KB. This buffer is used for:

- Combining cache blocks with adjacent disk block numbers, in order to write them to disk in groups, and save on latency and overhead
- Reading ahead a group of blocks, and then converting them to normal cache blocks.

Because there is significant overhead involved in accessing the disk drive, read-ahead improves performance significantly by reading groups of blocks at once.

TUNABLE PARAMETERS

There are certain operational parameters that control the disk cache operation which are tunable. A number of *preset* parameter sets is provided, dependent on the size of the cache. These should suffice for most purposes, but under certain types of workload, it may be desirable to tune these parameters to better suite the particular workload patterns.

See **dcacheDevTune()** for description of the tunable parameters. It is recommended to call **dcacheShow()** after calling **dcacheTune()** in order to verify that the parameters were set as requested, and to inspect the cache statistics which may change dramatically. Note that the hit ratio is a principal indicator of cache efficiency, and should be inspected during such tuning.

BACKGROUND UPDATING

A dedicated task will be created to take care of updating the disk with blocks that have been modified in cache. The time period between updates is controlled with the tunable parameter *syncInterval*. Its priority should be set above the priority of any CPU-bound tasks so as to assure it can wake up frequently enough to keep the disk synchronized with the cache. There is only one such task for all cache devices configured. The task name is *tDcacheUpd*

The updating task also has the responsibility to invalidate disk cache blocks for removable devices which have not been used for 2 seconds or more.

There are a few global variables which control the parameters of this task, namely:

dcacheUpdTaskPriority

controls the default priority of the update task, and is set by default to 250.

dcacheUpdTaskStack

is used to set the update task stack size.

dcacheUpdTaskOptions

controls the task options for the update task.

All the above global parameters must be set prior to calling **dcacheDevCreate()** for the first time, with the exception of *dcacheUpdTaskPriority*, which may be modified in run-time, and takes effect almost immediately. It should be noted that this priority is not entirely fixed, at times when critical disk operations are performed, and **FIOFLUSH** ioctl is called, the caller task will temporarily *loan* its priority to the update task, to insure the completion of the flushing operation.

REMOVABLE DEVICES

For removable devices, disk cache provides these additional features:

disk updating

is performed such that modified blocks will be written to disk within one second, so as to minimize the risk of losing data in case of a failure or disk removal.

error handling

includes a test for disk removal, so that if a disk is removed from the drive while an I/O operation is in progress, the disk removal event will be set immediately.

disk signature

which is a checksum of the disk's boot block, is maintained by the cache control structure, and it will be verified against the disk if it was idle for 2 seconds or more. Hence if during that idle time a disk was replaced, the change will be detected on the next disk access, and the condition will be flagged to the file system.

NOTE

It is very important that removable disks should all have a unique volume label, or volume serial number, which are stored in the disk's boot sector during formatting. Changing disks which have an identical boot sector may result in failure to detect the change, resulting in unpredictable behavior, possible file system corruption.

CACHE IMPLEMENTATION

Most Recently Used (MRU) disk blocks are stored in a collection of memory buffers called the disk cache. The purpose of the disk cache is to reduce the number of disk accesses and to accelerate disk read and write operations, by means of the following techniques:

- Most Recently Used blocks are stored in RAM, which results in the most frequently accessed data being retrieved from memory rather than from disk.
- Reading data from disk is performed in large units, relying on the read-ahead feature, one of the disk cache's tunable parameters.
- Write operations are optimized because they occur to memory first. Then updating the disk happens in an orderly manner, by delayed write, another tunable parameter.

Overall, the main performance advantage arises from a dramatic reduction in the amount of time spent by the disk drive seeking, thus maximizing the time available for the disk to read and write actual data. In other words, you get efficient use of the disk drive's available throughput. The disk cache offers a number of operational parameters that can be tuned by the user to suit a particular file system workload pattern, for example, delayed write, read ahead, and bypass threshold.

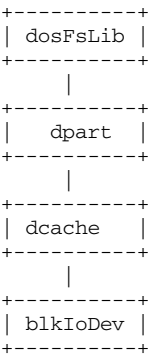
The technique of delaying writes to disk means that if the system is turned off unexpectedly, updates that have not yet been written to the disk are lost. To minimize the effect of a possible crash, the disk cache periodically updates the disk. Modified blocks of data are not kept in memory more than a specified period of time. By specifying a small update period, the possible worst-case loss of data from a crash is the sum of changes possible during that specified period. For example, it is assumed that an update period of 2 seconds is sufficiently large to effectively optimize disk writes, yet small enough to make the potential loss of data a reasonably minor concern. It is possible to set the update period to 0, in which case, all updates are flushed to disk immediately. This is essentially the equivalent of using the **DOS_OPT_AUTOSYNC** option in earlier **dosFsLib** implementations. The disk cache allows you to negotiate between disk performance and memory consumption: The more memory allocated to the disk cache, the higher the "hit ratio" observed, which means

increasingly better performance of file system operations. Another tunable parameter is the bypass threshold, which defines how much data constitutes a request large enough to justify bypassing the disk cache. When significantly large read or write requests are made by the application, the disk cache is circumvented and there is a direct transfer of data between the disk controller and the user data buffer. The use of bypassing, in conjunction with support for contiguous file allocation and access (via the FIOCONTIG `ioctl()` command and the `DOS_O_CONTIG open()` flag), should provide performance equivalent to that offered by the raw file system (rawFs).

PARTITION INTERACTION

The dcache CBIO layer is intended to operate atop an entire fixed disk device. When using the dcache layer with the dpart CBIO partition layer, it is important to place the dcache layer below the partition layer.

For example:



ENABLE/DISABLE THE DISK CACHE

The function `dcacheDevEnable` is used to enable the disk cache. The function `dcacheDevDisable` is used to disable the disk cache. When the disk cache is disabled, all IO will bypass the cache layer.

INCLUDE FILES	none
SEE ALSO	dosFsLib, cbioLib, dpartCbio

dirLib

NAME	dirLib – directory handling library (POSIX)
------	---

ROUTINES	opendir() – open a directory for searching (POSIX) readdir() – read one entry from a directory (POSIX) readdir_r() – read one entry from a directory (POSIX) rewinddir() – reset position to the start of a directory (POSIX) closedir() – close a directory (POSIX) stat() – get file status information (POSIX) stat() – get file status information using a pathname (POSIX) fstatfs() – get file status information (POSIX) statfs() – get file status information using a pathname (POSIX) utime() – update time on a file
DESCRIPTION	This library provides POSIX-defined routines for opening, reading, and closing directories on a file system. It also provides routines to obtain more detailed information on a file or directory.
CONFIGURATION	To use the POSIX directory-handling library, configure VxWorks with the INCLUDE_POSIX_DIRLIB component.

SEARCHING DIRECTORIES

Basic directory operations, including **opendir()**, **readdir()**, **rewinddir()**, and **closedir()**, determine the names of files and subdirectories in a directory.

A directory is opened for reading using **opendir()**, specifying the name of the directory to be opened. The **opendir()** call returns a pointer to a directory descriptor, which identifies a directory stream. The stream is initially positioned at the first entry in the directory.

Once a directory stream is opened, **readdir()** is used to obtain individual entries from it. Each call to **readdir()** returns one directory entry, in sequence from the start of the directory. The **readdir()** routine returns a pointer to a **dirent** structure, which contains the name of the file (or subdirectory) in the **d_name** field.

The **rewinddir()** routine resets the directory stream to the start of the directory. After **rewinddir()** has been called, the next **readdir()** will cause the current directory state to be read in, just as if a new **opendir()** had occurred. The first entry in the directory will be returned by the first **readdir()**.

The directory stream is closed by calling **closedir()**.

GETTING FILE INFORMATION

The directory stream operations described above provide a mechanism to determine the names of the entries in a directory, but they do not provide any other information about those entries. More detailed information is provided by **stat()** and **fstat()**.

The **stat()** and **fstat()** routines are essentially the same, except for how the file is specified. The **stat()** routine takes the name of the file as an input parameter, while **fstat()** takes a file descriptor number as returned by **open()** or **creat()**. Both routines place the information from a directory entry in a **stat** structure whose address is passed as an input parameter. This structure is defined in the include file **stat.h**. The fields in the structure include the file

size, modification date/time, whether it is a directory or regular file, and various other values.

The **st_mode** field contains the file type; several macro functions are provided to test the type easily. These macros operate on the **st_mode** field and evaluate to **TRUE** or **FALSE** depending on whether the file is a specific type. The macro names are:

S_ISREG
test if the file is a regular file

S_ISDIR
test if the file is a directory

S_ISCHR
test if the file is a character special file

S_ISBLK
test if the file is a block special file

S_ISFIFO
test if the file is a FIFO special file

Only the regular file and directory types are used for VxWorks local file systems. However, the other file types may appear when getting file status from a remote file system (using NFS).

As an example, the **S_ISDIR** macro tests whether a particular entry describes a directory. It is used as follows:

```
char      *filename;
struct stat fileStat;

stat (filename, &fileStat);

if (S_ISDIR (fileStat.st_mode))
printf ("%s is a directory.\n", filename);
else
printf ("%s is not a directory.\n", filename);
```

See the **ls()** routine in **usrLib** for an illustration of how to combine the directory stream operations with the **stat()** routine.

INCLUDE FILES **dirent.h, stat.h**

dosFsCacheLib

NAME **dosFsCacheLib** – MS-DOS media-compatible Cache library

ROUTINES **dosFsCacheLibInit()** – initialize dosFsCache library.

dosFsDefaultDataCacheSizeGet() – get the default data cache size
dosFsDefaultDirCacheSizeGet() – get the default directory cache size
dosFsDefaultFatCacheSizeGet() – get the default FAT cache size
dosFsDefaultCacheSizeSet() – set the default disk cache size
dosFsCacheOptionsSet() – set this dosFs volume's disk cache options
dosFsCacheOptionsGet() – get this dosFs volume's disk cache options
dosFsCacheCreate() – create cache for a DosFS volume
dosFsCacheDelete() – delete the disk cache for a dosFs volume
dosFsCacheTune() – tune a cache's settings
dosFsCacheInfo() – retrieve a cache's settings

DESCRIPTION This library implements a disk cache mechanism for the VxWorks MS-DOS compatible file system.

Disk cache is created on a per volume basis. This cache is used as a read/write cache for the File Allocation Table, directory entries, and data blocks.

To have dosFs cache support on a VxWorks image, the component **INCLUDE_DOSFS_CACHE** must be included. Also, the parameters **DOSFS_DEFAULT_DATA_CACHE_SIZE**, **DOSFS_DEFAULT_DIR_CACHE_SIZE** and **DOSFS_DEFAULT_FAT_CACHE_SIZE** should be set to appropriate values. Note that the values of these parameters will be used when automatically creating a disk cache for all the dosFs volumes in the system.

This automatic cache creation is done by the file system monitor (please refer to the FS monitor documentation for details), either at boot time or whenever a storage device with a MS-DOS compatible file system is detected, like for example when a FAT formatted floppy disk or USB device is inserted and accessed for the first time. If this behavior is not desired, the **DOSFS_DEFAULT_DATA_CACHE_SIZE**, **DOSFS_DEFAULT_DIR_CACHE_SIZE** and **DOSFS_DEFAULT_FAT_CACHE_SIZE** parameters can then be set to zero, or alternatively the cache for a specific volume can be removed using the API **dosFsCacheDelete()**, and a disk cache for every dosFs instantiation can be created manually using the API **dosFsCacheCreate()**. The value of this parameter can be get/set at runtime using the APIs **dosFsDefaultDataCacheSizeGet()**, **dosFsDefaultDirCacheSizeGet()**, **dosFsDefaultFatCacheSizeGet()**.

The **dosFsCacheShow()** routine gives a description of an specific cache in terms of its size, its current allocation status, and its hit/miss ratio.

DISK CACHE ALGORITHM

The disk cache is composed internally of a number cache blocks, of the same size as the disk physical block (sector). These cache blocks are maintained in a list in "Most Recently Used" order, that is, blocks which are used are moved to the top of this list. When a block needs to be relinquished, and made available to contain a new disk block, the Least Recently Used block will be used for this purpose.

INCLUDE FILES **dosFsLib.h**

dosFsFmtLib

NAME	dosFsFmtLib – MS-DOS media-compatible file system formatting library
ROUTINES	dosFsVolFormat() – format an MS-DOS compatible volume dosFsVolFormatFd() – format an MS-DOS compatible volume via an opened FD dosFsFmtLibInit() – initialize the MS-DOS formatting library dosFsFmtTest() – UNITEST CODE
DESCRIPTION	<p>This module is a scaleable companion module for dosFsLib, and is intended to facilitate high level formatting of disk volumes.</p> <p>Calling dosFsVolFormat() routine allows complete control over the format used, parameters and allows one to supply a hook routine which could, for instance, interactively prompt the user to modify disk parameters.</p>
AVAILABILITY	<p>This routine is an optional part of the MS-DOS file system, and may be included in a target system if it is required to be able to format new volumes.</p> <p>In order to include this option, the following function needs to be invoked during system initialization:</p> <pre>void dosFsFmtLibInit(void);</pre> <p>See reference page dosFsVolFormat() for complete description of supported formats, options and arguments.</p>
INCLUDE FILES	none
SEE ALSO	dosFsLib

dosFsLib

NAME	dosFsLib – MS-DOS media-compatible file system library
ROUTINES	dosfsHostToDisk32() – convert uint32_t from host to on-disk format dosfsHostToDisk16() – convert uint16_t from host to on-disk format dosfsDiskToHost32() – convert uint32_t from on-disk to host format dosfsDiskToHost16() – convert uint16_t from on-disk to host format dosFsVolumeOptionsSet() – set this volume's disk options dosFsVolumeOptionsGet() – get this volume's disk options dosSetVolCaseSens() – set case sensitivity of volume dosFsVolDescGet() – convert a device name into a DOS volume descriptor pointer.

dosFsVolUnmount() – unmount a dosFs volume
dosFsChkDsk() – make volume integrity checking.
dosFsVollsFat12() – determine if a MSDOS volume is FAT12 or FAT16
dosFsFdFree() – free a file descriptor
dosFsFdGet() – get an available file descriptor
dosPathParse() – parse a full pathname into an array of names.
dosFsOpen() – open a file on a dosFs volume
dosFsClose() – close a dosFs file
dosFsIoctl() – do device specific control function
dosFsLastAccessDateEnable() – enable last access date updating for this volume
dosFsLibInit() – prepare to use the dosFs library
dosFsDevCreate() – create file system device.
dosFsDevDelete() – delete a dosFs volume
dosFsMonitorDevCreate() – create a dosFs volume through the fs monitor
dosFsDiskProbe() – probe if a device contains a valid dosFs
dosFsHdlrInstall() – install handler.
dosFsXbdBlkRead() – read blocks from the underlying XBD block device.
dosFsXbdBlkWrite() – write blocks to the underlying XBD block device.
dosFsXbdBytesRW() – read/write bytes to/from the underlying XBD block device.
dosFsXbdBlkCopy() – copy blocks on the underlying XBD block device.
dosFsXbdIoctl() – Misc control operations

DESCRIPTION This library implements the MS-DOS compatible file system. This is a multi-module library, which depends on sub-modules to perform certain parts of the file system functionality. A number of different file system format variations are supported.

USING THIS LIBRARY

The various routines provided by the VxWorks DOS file system (dosFs) may be separated into three broad groups: general initialization, device initialization, and file system operation.

The **dosFsLibInit()** routine is the principal initialization function; it should be called once during system initialization, regardless of how many dosFs devices are to be used.

Another dosFs routine is used for device initialization. For each dosFs device, **dosFsDevCreate()** must be called to install the device in VxWorks device list. In the case where partitioned disks are used, **dosFsDevCreate()** must be called for each partition that is anticipated, thereby it is associated with a logical device name, so it can be later accessed via the I/O system. Note that starting from VxWorks 6.2, the job of instantiating file systems is done automatically by the File System Monitor module, either at boot time or whenever removable media is inserted in the system (such as a floppy disk or a USB device). Please refer to the File System Monitor documentation for further details.

In case of a removable media, device access and file system instantiation will be done only when the logical device is first accessed by the application.

More detailed information on all of these routines is provided below.

INITIALIZING DOSFSLIB

To enable this file system in a particular VxWorks configuration, a library initialization routine must be called for each sub-module of the file system, as well as for the underlying disk cache, partition manager and drivers. This is usually done at system initialization time, within the *usrRoot* task context.

Following is the list of initialization routines that need to be called:

dosFsLibInit

(mandatory) initialize the principle dosFs module. Must be called first.

dosFsFatInit

(mandatory) initialize the File Allocation Table handler, which supports 12-bit, 16-bit and 32-bit FATs.

dosVDirLibInit

(choice) install the variable size directory handler supporting Windows-compatible Long File Names (VFAT) Directory Handler.

dosDirOldLibInit

(choice) install the fixed size directory handler which supports read-only access to old-fashioned 8.3 MS-DOS file names, and Wind River Systems proprietary long file names (VXLONG).

dosFsFmtLibInit

(optional) install the volume formatting module.

dosChkLibInit

(optional) install the file system consistency checking module.

dosFsCacheLibInit

(optional) install the file system cacheing module.

The two Directory handlers which are marked *choice* are installed in accordance with the system requirements, either one of these modules could be installed or both, in which case the VFAT will take precedence for MS-DOS compatible volumes.

DEFINING A DOSFS DEVICE

The **dosFsDevCreate()** routine associates a device with the **dosFsLib** functions. It expects four parameters:

- (1) A pointer to a name string, to be used to identify the device - logical device name. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the **iosDevShow()** routine.
- (2) *device_t* - a XBD for the device on which to create the file system. It could be a partition XBD, an XBD block wrapper, or an ATA device XBD for example.
- (3) A maximum number of files can be simultaneously opened on a particular device.

- (4) Flags for volume checking, metadata integrity, and file name interpretation. Because volume integrity check utility can be automatically invoked every time a device is mounted, this parameter indicates whether the consistency check needs to be performed automatically on a given device, and on what level of verbosity is required. In any event, the consistency check may be invoked at a later time e.g. by calling **chkdsk()**. See description for FIOCHKDSK ioctl command for more information.

For example:

```
dosFsDevCreate
(
    "/sd0",          /* name to be used for volume */
    device,          /* underlying XBD device */
    10,              /* max no. of simultaneously open files */
    DOS_CHK_REPAIR | DOS_CHK_VERB_1
    /* check volume during mounting and repair */
    /* errors, and display volume statistics */
)
```

Once **dosFsDevCreate()** has been called, the device can be accessed using *ioLib* generic I/O routines: **open()**, **read()**, **write()**, **close()**, **ioctl()**, **remove()**. Also, the user-level utility functions may be used to access the device at a higher level (See **usrFsLib** reference page for more details).

DEVICE AND PATH NAMES

On true MS-DOS machines, disk device names are typically of the form "A:", that is, a single letter designator followed by a colon. Such names may be used with the VxWorks dosFs file system. However, it is possible (and desirable) to use longer, more mnemonic device names, such as "DOS1:", or "/floppy0". The name is specified during the **dosFsDevCreate()** call. Since most of the time the call to this routine is done automatically by the File System Monitor module, **fsmNameInstall()** can be called previously to specify the desired name for the device. Please refer to the fsMonitor documentation for further details.

The pathnames used to specify dosFs files and directories may use either forward slashes ("/") or backslashes ("\") as separators. These may be freely mixed. The choice of forward slashes or backslashes has absolutely no effect on the directory data written to the disk. (Note, however, that forward slashes are not allowed within VxWorks dosFs filenames, although they are normally legal for pure MS-DOS implementations.)

Use of forward slashes ("/") is recommended at all times.

The leading slash of a dosFs pathname following the device name is optional. For example, both "DOS1:newfile.new" and "DOS1:/newfile.new" refer to the same file.

USING EXTENDED DIRECTORY STRUCTURE

This library supports DOS4.0 standard file names which fit the restrictions of eight upper-case characters optionally followed by a three-character extension, as well as Windows style VFAT standard long file names that are stored mixed cased on disk, but are case insensitive when searched and matched (e.g. during **open()** call). The VFAT long file

name is stored in a variable number of consecutive directory entries. Both standards restrict file size to 4 GB (32 bit value).

To provide additional flexibility, this implementation of the DOS file system provides proprietary long file name format (VXLONGNAMES), which uses a simpler directory structure: the directory entry is of fixed size. When this option is used, file names may consist of any sequence of up to 40 ASCII characters. No case conversion is performed, and file name match is case-sensitive. With this directory format the file maximum size is expanded to 1 Terabyte (40 bit value). This option only supports read-only access to files in the VxWorks 6.2 version though.

NOTE

Because special directory entries are used on the disk, disks which use the extended names are *not* compatible with other implementation of the MS-DOS systems, and cannot be read on MS-DOS or Windows machines.

To enable the extended file names, set the `DOS_OPT_VXLONGNAMES` flag when calling `dosFsVolFormat()`.

USING UNICODE CHARACTERS

When Unicode characters are in use, they are encoded in UTF-8 through the `open()` and `readdir()` interface, and in Windows-compatible UTF-16 format on-disk. The translation between external (UTF-8) and internal (UTF-16) encodings is automatic, avoiding all the byte-order problems associated with UTF-16 encodings.

Existing VxWorks file systems that use "high bit" characters (such as ISO Latin 1 character sets) are not compatible with Unicode encodings. For this reason, Unicode file names must currently be enabled explicitly using the `DOS_FILENAMES_UNICODE` flag.

Unicode is only supported on VFAT (variable-length file name) volumes.

READING DIRECTORY ENTRIES

Directories on VxWorks dosFs volumes may be searched using the `opendir()`, `readdir()`, `rewinddir()`, and `closedir()` routines. These calls allow the names of files and subdirectories to be determined.

To obtain more detailed information about a specific file, use the `fstat()` or `stat()` routine. Along with standard file information, the structure used by these routines also returns the file attribute byte from a dosFs directory entry.

For more information, see the manual entry for `dirLib`.

SYNCHRONOUS FILES

Files can be opened with the `O_SYNC` flag, indicating that each write should be immediately written to the backing media. This includes synchronizing the FAT and the directory entries.

FILE DATE AND TIME

Directory entries on dosFs volumes contain creation, last modification time and date, and the last access date for each file or subdirectory. Directory last modification time and date fields are set only when a new entry is created, but not when any directory entries are deleted. The last access date field indicates the date of the last read or write. The last access date field is an optional field, per Microsoft. By default, file open-read-close operations do not update the last access date field. This default avoids media writes (writing out the date field) during read only operations. In order to enable the updating of the optional last access date field for open-read-close operations, you must call

dosFsLastAccessDateEnable(), passing it the volumes **DOS_VOLUME_DESC_ID** and **TRUE**.

The dosFs file system uses the ANSI **time()** function, that returns system clock value to obtain date and time. It is recommended that the target system should set the system time during system initialization time from a network server or from an embedded Calendar / Clock hardware component, so that all files on the file system would be associated with a correct date and time.

The file system consistency checker (see below) sets system clock to value following the latest date-time field stored on the disk, if it discovers, that function **time()** returns a date earlier then Jan 1, 1998, meaning that the target system does not have a source of valid date and time to synchronize with.

See also the reference manual entry for **ansiTime**.

FILE ATTRIBUTES

Directory entries on dosFs volumes contain an attribute byte consisting of bit-flags which specify various characteristics of the entry. The attributes which are identified are: read-only file, hidden file, system file, volume label, directory, and archive. The VxWorks symbols for these attribute bit-flags are:

DOS_ATTR_RDONLY

File is write-protected, can not be modified or deleted.

DOS_ATTR_HIDDEN

this attribute is not used by VxWorks.

DOS_ATTR_SYSTEM

this attribute is not used by VxWorks.

DOS_ATTR_VOL_LABEL

directory entry describes a volume label, this attribute can not be set or used directly, see **ioctl()** command **FIOLABELGET** and **FIOLABELSET** below for volume label manipulation.

DOS_ATTR_DIRECTORY

directory entry is a subdirectory, this attribute can not be set directly.

DOS_ATTR_ARCHIVE

this attribute is not used by VxWorks.

All the flags in the attribute byte, except the directory and volume label flags, may be set or cleared using the **ioctl()** FIOATTRIBSET function. This function is called after opening the specific file whose attributes are to be changed. The attribute byte value specified in the FIOATTRIBSET call is copied directly. To preserve existing flag settings, the current attributes should first be determined via **fstat()**, and the appropriate flag(s) changed using bitwise AND or OR operations. For example, to make a file read-only, while leaving other attributes intact:

```
struct stat fileStat;

fd = open ("file", O_RDONLY, 0);      /* open file          */
fstat (fd, &fileStat);               /* get file status    */

ioctl (fd, FIOATTRIBSET, (fileStat.st_attr | DOS_ATTR_RDONLY));
/* set read-only flag */
close (fd);                          /* close file         */
```

See also the reference manual entry for **attrib()** and **xattrib()** for user-level utility routines which control the attributes of files or file hierarchy.

CONTIGUOUS FILE SUPPORT

The VxWorks dosFs file system provides efficient files storage: space will be allocated in groups of clusters (also termed *extents*) so that a file will be composed of relatively large contiguous units. This nearly contiguous allocation technique is designed to effectively eliminate the effects of disk space fragmentation, keeping throughput very close to the maximum of which the hardware is capable.

However dosFs provides mechanism to allocate truly contiguous files, meaning files which are made up of a consecutive series of disk sectors. This support includes both the ability to allocate contiguous space to a file and optimized access to such a file when it is used. Usually this will somewhat improve performance when compared to Nearly Contiguous allocation, at the price of disk space fragmentation.

To allocate a contiguous area to a file, the file is first created in the normal fashion, using **open()** or **creat()**. The file descriptor returned during the creation of the file is then used to make an **ioctl()** call, specifying the FIOCONTIG or FIOCONTIG64 function. The last parameter to the FIOCONTIG function is the size of the requested contiguous area in bytes. If the FIOCONTIG64 is used, the last parameter is pointer to 64-bit integer variable, which contains the required file size. It is also possible to request that the largest contiguous free area on the disk be obtained. In this case, the size value **CONTIG_MAX (-1)** is used instead of an actual size. These **ioctl()** codes are not supported for directories. The volume is searched for a contiguous area of free space, which is assigned to the file. If a segment of contiguous free space large enough for the request was not found, **ERROR** is returned, with *errno* set to **S_dosFsLib_NO_CONTIG_SPACE**.

When contiguous space is allocated to a file, the file remains empty, while the newly allocated space has not been initialized. The data should be then written to the file, and eventually, when all data has been written, the file is closed. When file is closed, its space is truncated to reflect the amount of data actually written to the file. This file may then be again

opened and used for further I/O operations **read()** or **write()**, but it can not be guaranteed that appended data will be contiguous to the initially written data segment.

For example, the following will create a file and allocate 85 Mbytes of contiguous space:

```
fd = creat ("file", O_RDWR, 0);           /* open file          */
status = ioctl (fd, FIOCONTIG, 85*0x100000); /* get contiguous area */
if (status != OK)
    ...                                   /* do error handling   */
close (fd);                             /* close file          */
```

In contrast, the following example will create a file and allocate the largest contiguous area on the disk to it:

```
fd = creat ("file", O_RDWR, 0);           /* open file          */
status = ioctl (fd, FIOCONTIG, CONTIG_MAX); /* get contiguous area */
if (status != OK)
    ...                                   /* do error handling   */
close (fd);                             /* close file          */
```

NOTE

the FIOCONTIG operation should take place right after the file has been created, before any data is written to the file. Directories may not be allocated a contiguous disk area.

To determine the actual amount of contiguous space obtained when **CONTIG_MAX** is specified as the size, use **fstat()** to examine the number of blocks and block size for the file.

When any file is opened, it may be checked for contiguity. Use the extended flag **DOS_O_CONTIG_CHK** when calling **open()** to access an existing file which may have been allocated contiguous space. If a file is detected as contiguous, all subsequent operations on the file will not require access to the File Allocation Table, thus eliminating any disk Seek operations. The down side however is that if this option is used, **open()** will take an amount of time which is linearly proportional of the file size.

CHANGING, UNMOUNTING, AND SYNCHRONIZING DISKS

Buffering of disk data in RAM, and synchronization of these buffers with the disk are handled by the disk cache. See reference manual on **dosFsCacheLib** for more details. Detection of removable disk replacement is done by the File System Monitor subsystem.

If a disk is physically removed, the File System Monitor subsystem will delete the filesystem entry from coreIO and free all its allocated resources, including disk cache buffers.

If a new DOS FS formatted disk is inserted, it will be detected by the File System Monitor subsystem and a DOS FS filesystem will be automatically created with the name previously registered through a call to **fsmNameInstall()** (or a default name will be assigned), and with the global parameters **DOSFS_DEFAULT_MAX_FILES**, **DOSFS_DEFAULT_CREATE_OPTIONS**, and if disk cache is supported (see **dosFsCacheLib** for details), with a **DOSFS_DEFAULT_CACHE_SIZE** cache.

IOCTL FUNCTIONS

The dosFs file system supports the following **ioctl()** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions may be any file descriptor which is opened to a file or directory on the volume or to the volume itself. There are some **ioctl()** commands, that expect a 32-bit integer result (FIONFREE, FIOWHERE, etc.). However, disks and files which are greater than 4GB are supported. In order to solve this problem, new **ioctl()** functions have been added to support 64-bit integer results. They have the same name as basic functions, but with suffix *64*, namely: FIONFREE64, FIOWHERE64 and so on. These commands expect a pointer to a 64-bit integer, i.e.:

```
long long *arg ;
```

as the 3rd argument to the **ioctl()** function. If a value which is requested with a 32-bit **ioctl()** command is too large to be represented in the 32-bit variable, **ioctl()** will return **ERROR**, and *errno* will be set to **S_dosFsLib_32BIT_OVERFLOW**.

FIOUNMOUNT

Unmounts a disk volume. It performs the same function as **dosFsVolUnmount()**. This function must not be called from interrupt level:

```
status = ioctl (fd, FIOUNMOUNT, 0);
```

FIOGETNAME

Gets the file name of the file descriptor and copies it to the buffer *nameBuf*. Note that *nameBuf* must be large enough to contain the largest possible path name.

```
status = ioctl (fd, FIOGETNAME, &nameBuf );
```

FIORENAME

Renames the file or directory to the string *newname*:

```
fd = open( "oldname", O_RDONLY, 0 );  
status = ioctl (fd, FIORENAME, "newname");
```

FIOUPDATE

Updates the dosFs create options to the new value *newoptions*

```
int newOptions;  
status = ioctl (fd, FIOUPDATE, newOptions);
```

FIOMOVE

Moves the file or directory to the string *newname*:

```
fd = open( "oldname", O_RDONLY, 0 );  
status = ioctl (fd, FIOMOVE, "newname");
```

FIOSEEK

Sets the current byte offset in the file to the position specified by *newOffset*. This function supports offsets in 32-bit value range. Use FIOSEEK64 for larger position values:

```
status = ioctl (fd, FIOSEEK, newOffset);
```

FIOSEEK64

Sets the current byte offset in the file to the position specified by *newOffset*. This function supports offsets in 64-bit value range:

```
long long    newOffset;  
status = ioctl (fd, FIOSEEK64, (int) & newOffset);
```

FIOWHERE

Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. This function returns a 32-bit value. It takes no additional argument:

```
position = ioctl (fd, FIOWHERE, 0);
```

FIOWHERE64

Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. This function returns a 64-bit value in *position*:

```
long long    position;  
status = ioctl (fd, FIOWHERE64, (int) & position);
```

FIOFLUSH

Flushes disk cache buffers. It guarantees that any output that has been requested is actually written to the device:

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOSYNC

Updates the FAT copy for the passed file descriptor, then flushes and invalidates the dosFs cache buffers for the file descriptor's volume. FIOSYNC ensures that any outstanding output requests for the passed file descriptor are written to the device and a subsequent I/O operation will fetch data directly from the physical medium. To safely sync a volume for shutdown, all open file descriptor's should at the least be FIOSYNC'd by the application. Better, all open FD's should be closed by the application and the volume should be unmounted via FIOUNMOUNT.

```
status = ioctl (fd, FIOSYNC, 0);
```

FIOTRUNC

Sets the specified file's length to *newLength* bytes. Any disk clusters which had been allocated to the file but are now unused are deallocated while additional clusters are zeroed, and the directory entry for the file is updated to reflect the new length. Only regular files may be truncated; attempts to use FIOTRUNC on directories will return an error.

```
status = ioctl (fd, FIOTRUNC, newLength);
```

FIOTRUNC64

Similar to FIOTRUNC, but can be used for files larger, than 4GB.

```
long long newLength = .....;  
status = ioctl (fd, FIOTRUNC, (int) & newLength);
```

FIONREAD

Copies to *unreadCount* the number of unread bytes in the file:

```
unsigned long unreadCount;
status = ioctl (fd, FIONREAD, &unreadCount);
```

FIONREAD64

Copies to *unreadCount* the number of unread bytes in the file. This function returns a 64-bit integer value:

```
long long unreadCount;
status = ioctl (fd, FIONREAD64, &unreadCount);
```

FIONFREE

Copies to *freeCount* the amount of free space, in bytes, on the volume:

```
unsigned long freeCount;
status = ioctl (fd, FIONFREE, &freeCount);
```

FIONFREE64

Copies to *freeCount* the amount of free space, in bytes, on the volume. This function can return value in 64-bit range:

```
long long freeCount;
status = ioctl (fd, FIONFREE64, &freeCount);
```

FIOMKDIR

Creates a new directory with the name specified as *dirName*:

```
status = ioctl (fd, FIOMKDIR, "dirName");
```

FIORMDIR

Removes the directory whose name is specified as *dirName*:

```
status = ioctl (fd, FIORMDIR, "dirName");
```

FIOLABELGET

Gets the volume label (located in root directory) and copies the string to *labelBuffer*. If the label contains **DOS_VOL_LABEL_LEN** significant characters, resulting string is not **NULL** terminated:

```
char    labelBuffer [DOS_VOL_LABEL_LEN];
status = ioctl (fd, FIOLABELGET, (int)labelBuffer);
```

FIOLABELSET

Sets the volume label to the string specified as *newLabel*. The string may consist of up to eleven ASCII characters:

```
status = ioctl (fd, FIOLABELSET, (int)"newLabel");
```

FIOATTRIBSET

Sets the file attribute byte in the DOS directory entry to the new value *newAttrib*. The file descriptor refers to the file whose entry is to be modified:

```
status = ioctl (fd, FIOATTRIBSET, newAttrib);
```

FIOCONTIG

Allocates contiguous disk space for a file or directory. The number of bytes of requested space is specified in *bytesRequested*. In general, contiguous space should be allocated immediately after the file is created:

```
status = ioctl (fd, FIOCONTIG, bytesRequested);
```

FIOCONTIG64

Allocates contiguous disk space for a file or directory. The number of bytes of requested space is specified in *bytesRequested*. In general, contiguous space should be allocated immediately after the file is created. This function accepts a 64-bit value:

```
long long bytesRequested;  
status = ioctl (fd, FIOCONTIG64, &bytesRequested);
```

FIONCONTIG

Copies to *maxContigBytes* the size of the largest contiguous free space, in bytes, on the volume:

```
status = ioctl (fd, FIONCONTIG, &maxContigBytes);
```

FIONCONTIG64

Copies to *maxContigBytes* the size of the largest contiguous free space, in bytes, on the volume. This function returns a 64-bit value:

```
long long maxContigBytes;  
status = ioctl (fd, FIONCONTIG64, &maxContigBytes);
```

FIOREADDIR

Reads the next directory entry. The argument *dirStruct* is a DIR directory descriptor. Normally, the **readdir()** routine is used to read a directory, rather than using the FIOREADDIR function directly. See **dirLib**.

```
DIR dirStruct;  
fd = open ("directory", O_RDONLY);  
status = ioctl (fd, FIOREADDIR, &dirStruct);
```

FIOFSTATGET

Gets file status information (directory entry data). The argument *statStruct* is a pointer to a stat structure that is filled with data describing the specified file. Normally, the **stat()** or **fstat()** routine is used to obtain file information, rather than using the FIOFSTATGET function directly. See **dirLib**.

```
struct stat statStruct;  
fd = open ("file", O_RDONLY);  
status = ioctl (fd, FIOFSTATGET, (int)&statStruct);
```

FIOTIMESET

Update time on a file. *arg* shall be a pointer to a utimbuf structure, see **utime.h**. If *arg* is value NULL, the current system time is used for both actime and modtime members. If *arg* is not NULL then the utimbuf structure members actime and modtime are used as passed. If actime is zero value, the file access time is not updated (the operation is ignored). If modtime is zero, the file modification time is not updated (the operation is ignored). See also **utime()**

```
struct utimbuf newTimeBuf;;  
newTimeBuf.modtime = newTimeBuf.actime = fileNewTime;  
fd = open ("file", O_RDONLY);  
status = ioctl (fd, FIOTIMESET, (int)&newTimeBuf);
```

FIOCHKDSK

This function invokes the integral consistency checking. During the test, the file system will be blocked from application code access, and will emit messages describing any inconsistencies found on the disk, as well as some statistics, depending on the verbosity level in the *flags* argument. Depending on the repair permission value in *flags* argument, the inconsistencies will be repaired, and changes written to disk or only reported. Argument *flags* should be composed of bitwise or-ed verbosity level value and repair permission value. Possible repair levels are:

DOS_CHK_ONLY (1)

Only report errors, do not modify disk.

DOS_CHK_REPAIR (2)

Repair any errors found.

Possible verbosity levels are:

DOS_CHK_VERB_SILENT (0xff00)

Do not emit any messages, except errors encountered.

DOS_CHK_VERB_1 (0x0100)

Display some volume statistics when done testing, as well

DOS_CHK_VERB_2 (0x0200)

In addition to the above option, display path of every file, while it is being checked. This option may significantly slow down the test process.

NOTE

In environments with reduced RAM size check disk uses reserved FAT copy as temporary buffer, it can cause respectively long time of execution on a slow CPU architectures..

See also the reference manual **usrFsLib** for the **chkdsk()** user level utility which may be used to invoke the **FIOCHKDSK ioctl()**. The volume root directory should be opened, and the resulting file descriptor should be used:

```
int fd = open (device_name, O_RDONLY, 0);
status = ioctl (fd, FIOCHKDSK, DOS_CHK_REPAIR | DOS_CHK_VERB_1);
close (fd);
```

Any other **ioctl()** function codes are passed to the underlying *XBD* modules for handling.

INCLUDE FILES **dosFsLib.h**

SEE ALSO **ioLib, iosLib, dirLib, usrFsLib, dosFsCacheLib, dosFsFmtLib, dosChkLib**, *Microsoft MS-DOS Programmer's Reference*, (Microsoft Press), *Advanced MS-DOS Programming*, (Ray Duncan, Microsoft Press), *VxWorks Programmer's Guide: I/O System, Local File Systems*

dosFsShow

NAME	dosFsShow – DosFS Show routines
ROUTINES	dosFsShow() – display dosFs volume configuration data. dosFsCacheShow() – show information regarding a dosFs volume's cache
DESCRIPTION	This library implements the DosFS Show routines.
INCLUDE FILES	none

dpartCbio

NAME	dpartCbio – generic disk partition manager
ROUTINES	dpartDevCreate() – Initialize a partitioned disk dpartPartGet() – retrieve handle for a partition
DESCRIPTION	<p>This module implements a generic partition manager using the CBIO API (see cbioLib) It supports creating a separate file system device for each of its partitions.</p> <p>This partition manager depends upon an external library to decode a particular disk partition table format, and report the resulting partition layout information back to this module. This module is responsible for maintaining the partition logic during operation.</p> <p>When using this module with the dcacheCbio module, it is recommended this module be the master CBIO device. This module should be above the cache CBIO module layer. This is because the cache layer is optimized to function efficiently atop a single physical disk drive. One should call dcacheDevCreate before dpartDevCreate.</p> <p>An implementation of the de-facto standard partition table format which is created by the MSDOS FDISK program is provided with the usrFdiskPartLib module, which should be used to handle PC-style partitioned hard or removable drives.</p>

EXAMPLE The following code will initialize a disk which is expected to have up to 4 partitions:

```
usrPartDiskFsInit( BLK_DEV * blkDevId )
{
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c", "/sd0d" };
    CBIO_DEV_ID cbioCache;
    CBIO_DEV_ID cbioParts;

    /* create a disk cache atop the entire BLK_DEV */
```



```

cbioCache = dcacheDevCreate ( blkDevId, NULL, 0, "/sd0" );

if (NULL == cbioCache)
{
    return (ERROR);
}

/* create a partition manager with a FDISK style decoder */

cbioParts = dpartDevCreate( cbioCache, 4, usrFdiskPartRead );

if (NULL == cbioParts)
{
    return (ERROR);
}

/* create file systems atop each partition */

dosFsDevCreate( devNames[0], dpartPartGet(cbioParts,0), 0x10, NONE);
dosFsDevCreate( devNames[1], dpartPartGet(cbioParts,1), 0x10, NONE);
dosFsDevCreate( devNames[2], dpartPartGet(cbioParts,2), 0x10, NONE);
dosFsDevCreate( devNames[3], dpartPartGet(cbioParts,3), 0x10, NONE);
}

```

Because this module complies with the CBIO programming interface on both its upper and lower layers, it is both an optional and a stackable module.

INCLUDE FILES none

SEE ALSO **dcacheLib**, **dosFsLib**, **usrFdiskPartLib**

dshmMuxLib

NAME dshmMuxLib – DSHM service/hardware bus multiplexer

ROUTINES

- dshmMuxLibInit()** – initialize the DSHM MUX
- dshmMuxHwRegister()** – register a hardware bus with the MUX
- dshmMuxHwGet()** – obtain an hardware registration handle based on name
- dshmMuxHwNodesNumGet()** – obtain the maximum number of nodes on a hardware bus
- dshmMuxHwTasGet()** – obtain the test-and-set routine on this bus
- dshmMuxHwTasClearGet()** – obtain the TAS clear routine on this bus
- dshmMuxHwOffToAddr()** – translate a shared memory offset to a local address
- dshmMuxHwAddrToOff()** – translate a local address to a shared memory offset
- dshmMuxHwLocalAddrGet()** – obtain address of the local node
- dshmMuxSvcNodeJoin()** – signal services that a node has joined the system
- dshmMuxSvcNodeLeave()** – signal services that a node has left the system
- dshmMuxSvcRegister()** – register a service with the MUX

dshMuxSvcObjGet() – retrieve a service object and protect it against deletion
dshMuxSvcObjRelease() – allows modifications to be made on a service object
dshMuxSvcWithdraw() – remove service from MUX
dshMuxWiddtdrawComplete() – signal service has finished withdrawing
dshMuxMsgSend() – transmit a message
dshMuxMsgRecv() – receive a message
dshMuxMemAlloc() – allocate shared memory from a specific hardware
dshMuxMemFree() – free allocated shared memory from a specific hardware

DESCRIPTION

The DSHM multiplexer allows for multiple hardware and services registration and thus usage of the DSHM system. Outgoing messages find the correct hardware bus on which to be transmitted and incoming messages are routed to the correct service for processing.

Registration

Both services and buses must register to be able to use the DSHM system. First, a bus register via a call to the **dshMuxHwRegister()** function, providing a set of callbacks routines for implementation of functionalities that are specific to it. This is performed in the hardware interface bring-up sequence. Buses must register by providing a unique string identifying them, normally specifying the bus type. Once the bus has registered, all other operations access its functionalities by providing a unique hardware identifier that has been linked with the bus at registration time. This identifier is obtained via a call to **dshMuxHwGet()**, by passing in the bus name used during registration.

When a bus has registered, services can then register to provide functionality on it. This is done through the **dshMuxSvcRegister()** call. The service must provide callback routines as well for handling of events directed to it, such as incoming message handling, nodes leaving and joining, and stopping the service. Once registered, the service then should obtain the bus identifier via a call to **dshMuxHwGet()**. From that point on, the service can make usage of the bus functionalities, such as sending messages to remote nodes.

Node joining

The services are responsible to track resources they use that are per-node. When a node joins, the **dshMuxSvcNodeJoin()** routine is called, calling all **join** callbacks provided by services registered on the bus. This callback can provide such functionalities such as sending a message to the remote node telling it the service is provided on this node, creating data structures that represent the remote node, allocating shared memory buffers to the remote node, etc.

Node leaving

As for the node joining event, services are responsible for handling of per-node resources they provide. When a node leaves, the **dshMuxSvcNodeLeave()** routine is called, which in turn calls all **leave** callbacks provided by services registered on the bus. These callbacks should normally take the reverse action the **join** callback performed.

Service leaving

A service can decide to leave willingly the DSHM system. If so, it must call the **dshmMuxSvcWithdraw()** routine. This places the service in a **quitting** state. While in this state, the service can perform different actions, such as telling the remote nodes that it does not provide its functionality anymore. This is left to the discretion of the service. All of it can be performed in the **stop** callback that can be registered with the service. When the service is satisfied that it has finished shutting down completely, it must then call the **dshmMuxSvcWithdrawComplete()** routine that removes the service registration from the MUX and allows another service with the same name to register in its place. This routine cannot be called from within the **stop** callback for various reasons.

Using the service

A service can provide to the MUX, at registration time, an object that contains all its pertinent information. When incoming messages intended for the service are received, a lock on the object is obtained. The service callback for the reception of messages must then release the lock when it has finished using the object. This asymmetry of having a different module acquiring and releasing the object allows the service to release the lock when deemed necessary rather than waiting to return control to the MUX to have that operation performed.

When making downcalls to the MUX, or simply using their object, services must obtain the lock via a call to the **dshmMuxSvcObjGet()** routine. The lock is release by a call to **dshmMuxSvcObjRelease()**. This prevents it from deletion.

Acquiring bus-specific information and functionalities

If needed by services, these can be acquired from the bus: atomic-set/clear routines, maximum number of nodes on the bus and address of local node on the bus. These functionalities can be called upon: local address to shared memory offset translation and vice-versa, as well as allocation and freeing of shared memory that is managed by the local node, on that particular bus.

Sending of messages

Messages must be built by declaring a message via the **DSHM()** macro. Then the header is built via the **DSHM_BUILD()** macro in **dshm.h**. The message body can be built via the **DSHM_DAT8_SET()**, **DSHM_DAT16_SET()** and **DSHM_DAT32_SET()** macros, in **dshm.h** as well. These take care of endiannes issues. If the message body is a byte stream, it can be set by accessing the data portion of the message with **DSHM_DAT_GET()**, and treating it as a byte array of size **DSHM_SIZE_DAT**. When the message is built, it is sent by a call to **dshmMuxMsgSend()**.

Receiving a message

When a message is received on the bus, the hardware interface calls **dshmMuxMsgRecv()**, which is responsible for calling the **rx** callback provided by the service the message is intended for. From that point on, the service is responsible for handling the message.

INCLUDE FILES none

SEE ALSO **dshm/dshm.h, dshm/dshmMuxLib.h**

dsiSockLib

NAME **dsiSockLib** – DSI sockets library

ROUTINES **dsiSysPoolShow()** – display DSI's system pool statistics
 dsiDataPoolShow() – display DSI's data pool statistics

ADDRESS FAMILY DSI sockets support only the **AF_LOCAL/AF_UNIX** Domain address family; use **AF_LOCAL/AF_UNIX** for the *domain* argument in subroutines that require it.

IOCTL FUNCTIONS
Sockets respond to the following **ioctl()** functions. These functions are defined in the header files **ioLib.h** and **ioctl.h**.

FIONBIO

Turns on/off non-blocking I/O.

```
on = TRUE;  
status = ioctl (sFd, FIONBIO, &on);
```

FIONREAD

Reports the number of bytes available to read on the socket. On the return of **ioctl()**, *bytesAvailable* has the number of bytes available to read on the socket.

```
status = ioctl (sFd, FIONREAD, &bytesAvailable);
```

INCLUDE FILES **dsiSockLib.h, un.h**

SEE ALSO **netLib, sockLib**, the VxWorks programmer's guides

edrErrLogLib

NAME **edrErrLogLib** – the ED&R error log library

ROUTINES **edrErrLogCreate()** – create a new log
 edrErrLogIterCreate() – create an iterator for traversing the log
 edrErrLogIterNext() – returns the next committed node
 edrErrLogAttach() – attach to an existing log

edrErrLogClear() – clear the log's contents
edrErrLogNodeCommit() – commits a previously allocated node
edrErrLogNodeAlloc() – allocate a node from the error log
edrErrLogNodeCount() – return the number of committed nodes in the log
edrErrLogMaxNodeCount() – return the maximum number of nodes in the log

DESCRIPTION

This library provides facilities for managing the error-log; the error-log is an integral part of the **edrLib()** library and as such it is primarily a private API but may be used to examine the ED&R log. The error-log acts as a ring buffer for a set of error-records and the minimum and maximum size of one node is fixed at creation time.

This library manipulates its internal data structures using only **intLock()** and **intUnlock()** to guarantee the integrity of the log. The log iterator uses a lock-free algorithm to iterate over the set of error-records thus allowing records to be added or removed while iterating over the log.

This library makes no use of any dynamically allocated memory.

CONSTRUCTION

An error-log is created by overlaying its structure onto an existing area of memory. To create a log you should call **edrErrLogCreate()** with the address and size of the memory region you have previously set aside. The area of memory could simply come from **malloc()**, or from a region reserved by **pmLib()**, for example.

For example, here's how to create a new log with 4K records and add 1 node to it.

```
void * pAddr = malloc (12000);
EDR_ERR_LOG * pLog = edrErrLogCreate (pAddr, 12000, 4096);

if (pLog != NULL)
{
    EDR_ERR_LOG_NODE * pNode;

    pNode = edrErrLogNodeAlloc (pLog);

    if (pNode != NULL)
    {
        memcpy (pNode->data, "foo", 3);
    }

    if (! edrErrLogNodeCommit (pLog, pNode))
        printf ("commit failed!\n");
}
```

ENUMERATION

The set of nodes can be accessed by creating a log iterator; **edrErrLogIterCreate()**. For each node retrieved the client should take a snapshot of the node's generation count, copy the payload to a separate buffer, and reevaluate that the generation-count didn't change during the copy. Using this approach it is possible to use multiple iterator-instances without requiring any mutual-exclusion barriers (or other locking primitives) leaving the log open to other concurrent writers. For example, the injection of errors from ISR routines, via **edrLib()**.

```

EDR_ERR_LOG_ITER iter;
EDR_ERR_LOG_NODE *pNode;
char *buf;

edrErrLogIterCreate (pLog, &iter, start, count);

if ((buf = malloc (pLog->header.payloadSize)) == NULL)
    return (ERROR);

while ((pNode = edrErrLogIterNext (&iter)) != NULL)
{
    int genCount = pNode->genCount;

    memcpy (buf, pNode->data, pLog->header.payloadSize);

    if (genCount == pNode->genCount)
    {
        /* The node wasn't changed asynchronously therefore
        @ its payload is still valid.
        */
    }
}

free (buf);

```

CONFIGURATION To use the EDR error log library, configure VxWorks with the **INCLUDE_EDR_ERRLOG** component.

INCLUDE **edrErrLogLib.h**

edrLib

NAME **edrLib** – Error Detection and Reporting subsystem

ROUTINES

- edrLibInit()** – initializes **edrLib**
- edrErrorInject()** – injects an error into the ED&R subsystem
- edrErrorLogClear()** – clears the ED&R error log
- edrErrorRecordCount()** – returns the number of error-records in the log
- edrErrorInjectHookAdd()** – adds a hook which gets called on error-injection
- edrErrorInjectHookDelete()** – removes an existing error-inject hook
- edrErrorInjectPrePostHookAdd()** – adds a hook which gets called before and after error-injection
- edrErrorInjectPrePostHookDelete()** – removes the existing pre/post hook
- edrErrorInjectTextHookAdd()** – adds a hook which gets called on record creation
- edrErrorInjectTextHookDelete()** – removes the existing text writing hook
- edrBootCountGet()** – returns the current boot count

DESCRIPTION

This library provides the public API for the ED&R subsystem, covering error injection, and the manipulation of error records within the error log.

It implements a circular log containing error-records (see struct **EDR_ERROR_RECORD** in file **edrLib.h**) that capture certain specific events within the VxWorks operating system. Each of these events is specifically instrumented with a call to **edrErrorInject()**, usually wrapped up in one of the macros in **edrLib.h** such as **EDR_KERNEL_FATAL_INJECT()** for fatal kernel-space errors.

The error-log is protected against being over-written by use of the MMU / **vmLib** facilities within VxWorks. However, from system initialisation to the time **edrLibInit()** is called, the log is not yet protected, and so does not need to be unprotected/re-protected. At all other times, the log is write-protected, except for the brief periods when an instance of **edrErrorInject()** is writing a record to the log.

This library uses the **edrErrLogLib** library to provide the implementation of the actual error log data structure. This implementation takes care of log integrity (w.r.t. interrupt locking, etc) but not memory protection -- that is handled by **edrLib** itself.

The information stored in the error record is dependent on some other VxWorks components to provide certain functionality for creating parts of the error-records. Specifically, the following components are required:

- **INCLUDE_EXC_SHOW** must be included to get a full detailed description of exception error-records.
- **INCLUDE_SHOW_ROUTINES** (or **INCLUDE_TASK_SHOW** in the project facility) must be included to get a full register dump from each error-record.
- **INCLUDE_DEBUG** must be included to get a code disassembly and traceback.

In the absence of these components, simple hex values for the information will be stored.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES

edrLib.h

edrShow

NAME	edrShow – ED&R Show Routines
ROUTINES	<p>edrErrorRecordDecode() – decode one error-record</p> <p>edrShow() – displays the ED&R error log to stdout</p> <p>edrFatalShow() – show all stored fatal type ED&R records</p> <p>edrInfoShow() – show all stored info type ED&R records</p> <p>edrIntShow() – show all stored interrupt type ED&R records</p> <p>edrInitShow() – show all stored init type ED&R records</p> <p>edrRebootShow() – show all stored reboot type ED&R records</p> <p>edrBootShow() – show all stored boot type ED&R records</p> <p>edrKernelShow() – show all stored kernel type ED&R records</p> <p>edrUserShow() – show all stored user type ED&R records</p> <p>edrRtpShow() – show all stored rtp type ED&R records</p> <p>edrClear() – a synonym for edrErrorLogClear</p> <p>edrInjectHookShow() – show the list of error injection hook routines</p> <p>edrInjectTextHookShow() – show the list of text injection hook routines</p> <p>edrInjectPrePostHookShow() – show the list of pre/post injection hook routines</p> <p>edrHookShow() – show the list of installed ED&R hook routines</p> <p>edrHelp() – prints helpful information on ED&R</p>
DESCRIPTION	<p>This module implements the show routines for the ED&R subsystem. The commands provided allow for displaying all or part of the stored ED&R log. It should be noted that not all error-record types have a complete ED&R record stored. For example, the BOOT and REBOOT records do not have a register set included in them and as a result the show routines will not display this information. All show commands will always display all stored information for a record.</p> <p>The function edrClear() will clear out the entire error-log, and should be used with utmost care. This is a destructive operation, and should be used sparingly, if at all.</p>
CONFIGURATION	To use the ED&R show routines, configure VxWorks with the INCLUDE_EDR_SHOW component.
INCLUDE FILES	edrLib.h

edrSysDbgLib

NAME	edrSysDbgLib – ED&R system-debug flag
------	---------------------------------------

ROUTINES	edrSystemDebugModeInit() – initialise the system mode debug flag edrSystemDebugModeGet() – indicates if the system is in debug mode edrSystemDebugModeSet() – modifies the system debug mode flag edrFlagsGet() – return the ED&R flags which are currently set edrIsDebugMode() – is the ED&R debug mode flag set?
DESCRIPTION	<p>This library provides access to the system debug flag.</p> <p>This flag indicates whether the system is in debug (also known as lab) mode, or in field (or deployed) mode.</p> <p>Certain system behaviours (see edrLib) are modified when in lab mode, specifically the system's response to exceptions in kernel and user mode. By default, the lab mode response is to put the failing component into a debuggable state, whereas the deployed mode response is to terminate (and attempt to restart) the failing component.</p>
CONFIGURATION	To use the ED&R system-debug flag, configure VxWorks with the INCLUDE_EDR_SYSDBG_FLAG component.
INCLUDE FILES	none

envLib

NAME	envLib – environment variable library
ROUTINES	envLibInit() – initialize environment variable facility envPrivateCreate() – create a private environment envPrivateDestroy() – destroy a private environment putenv() – set an environment variable getenv() – get an environment variable (ANSI) envShow() – display the environment for a task envGet() – return a pointer to the environment of a task
DESCRIPTION	<p>This library provides a UNIX-compatible environment variable facility. Environment variables are created or modified with a call to putenv():</p> <pre>putenv ("variableName=value");</pre> <p>The value of a variable may be retrieved with a call to getenv(), which returns a pointer to the value string.</p> <p>Tasks may share a common set of environment variables, or they may optionally create their own private environments, either automatically when the task create hook is installed, or by an explicit call to envPrivateCreate(). The task must be spawned with the VX_PRIVATE_ENV option set to receive a private set of environment variables. Private</p>

environments created by the task creation hook inherit the values of the environment of the task that called **taskSpawn()** (since task create hooks run in the context of the calling task).

INCLUDE FILES **envLib.h**

SEE ALSO UNIX BSD 4.3 manual entry for **environ(5V)**, *American National Standard for Information Systems -, Programming Language - C*, ANSI X3.159-1989: *General Utilities* (**stdlib.h**)

errnoLib

NAME **errnoLib** – error status library

ROUTINES **errnoGet()** – get the error status value of the calling task
errnoOfTaskGet() – get the error status value of a specified task
errnoSet() – set the error status value of the calling task
errnoOfTaskSet() – set the error status value of a specified task

DESCRIPTION This library contains routines for setting and examining the error status values of tasks and interrupts. Most VxWorks functions return **ERROR** when they detect an error, or **NULL** in the case of functions returning pointers. In addition, they set an error status that elaborates the nature of the error.

This facility is compatible with the UNIX error status mechanism in which error status values are set in the global variable **errno**. However, in VxWorks there are many task and interrupt contexts that share common memory space and therefore conflict in their use of this global variable. VxWorks resolves this in two ways:

- (1) For tasks, VxWorks maintains the **errno** value for each context separately. The value of **errno** for a task is stored in the task TCB. Regardless of task context code can always reference or modify **errno** directly, and only the currently executing task's value will be affected.
- (2) For interrupt service routines, VxWorks maintains a separate **errno**. Interrupt service routines can also reference or modify **errno** directly, and only the separate interrupt service routine value of **errno** will be affected.

The **errno** facility is used throughout VxWorks for error reporting. In situations where a lower-level routine has generated an error, by convention, higher-level routines propagate the same error status, leaving **errno** with the value set at the deepest level. Developers are encouraged to use the same mechanism for application modules where appropriate.

An error status is a 4-byte integer. By convention, the most significant two bytes are the module number, which indicates the module in which the error occurred. The lower two bytes indicate the specific error within that module. Module number 0 is reserved for UNIX error numbers so that values from the UNIX **errno.h** header file can be set and tested

without modification. Module numbers 1-500 decimal are reserved for VxWorks modules. These are defined in **vwModNum.h**. All other module numbers are available to applications.

VxWorks can include a special symbol table called **statSymTbl** which **printErrno()** uses to print human-readable error messages.

This table is created with the tool **makeStatTbl**, found in **host/hostOs/bin**. This tool reads all the **.h** files in a specified directory and generates a C-language file, which generates a symbol table when compiled. Each symbol consists of an error status value and its definition, which was obtained from the header file.

For example, suppose the header file **target/h/myFile.h** contains the line:

```
#define S_myFile_ERROR_TOO_MANY_COOKS      0x230003
```

The table **statSymTbl** is created by first running:

On UNIX:

```
makeStatTbl target/h > statTbl.c
```

On Windows:

```
makeStatTbl target/h
```

This creates a file **statTbl.c** in the current directory, which, when compiled, generates **statSymTbl**. The table is then linked with VxWorks. Normally, these steps are performed automatically by the makefile in **target/src/usr**.

If the user now types from the VxWorks shell:

```
-> printErrno 0x230003
```

The **printErrno()** routine would respond:

```
S_myFile_ERROR_TOO_MANY_COOKS
```

The **makeStatTbl** tool looks for error status lines of the form:

```
#define S_xxx    <n>
```

where *xxx* is any string, and *n* is any number. All VxWorks status lines are of the form:

```
#define S_thisFile_MEANINGFUL_ERROR_MESSAGE    0xnxxxx
```

where *thisFile* is the name of the module.

CONFIGURATION	This facility is always available without any additional configuration. To use it, add header files with status lines of the appropriate form and rebuild VxWorks.
INCLUDE FILES	errnoLib.h , The file vwModNum.h contains the module numbers for every VxWorks module., The include file for each module contains the error numbers which that module, can generate.
SEE ALSO	printErrno() , makeStatTbl

eventLib

NAME	eventLib – VxWorks events library
ROUTINES	eventReceive() – Wait for event(s) eventSend() – Send event(s) eventClear() – Clear the calling task's events register
DESCRIPTION	<p>Events are a means of communication and synchronization between tasks and interrupt service routines. Only tasks can receive events but both tasks and ISRs can send them. Events are an attractive lighter weight alternative to binary semaphores to perform task-to-task or ISR-to-task synchronization. The functionality provided by this library can be included/removed from the VxWorks kernel using the INCLUDE_VXEVENTS component.</p> <p>Events are similar to signals in that they are directed at one task and can be sent at any time regardless of the state of the said task. However they differ from signals in that the receiving task's execution is not altered by the arrival of events. The receiving task must explicitly check its event register to determine if it has received events.</p> <p>Each task has its own events register that can be filled by having tasks (even itself) and/or ISRs send events to the task. Events are generic in nature in that VxWorks does not assign specific meaning to any events. The parties communicating using events must have an a priori agreement on the meaning of individual events.</p> <p>Events are not accumulated. If the same event is received several times, it is treated as if it were received only once. It is not possible to track how many times each event has been received by a task.</p> <p>An event is actually a bit in a 32 bit word. Therefore up to 32 distinct events can be sent to a task.</p> <p>Semaphore and message queues can also send events automatically when they become available. For example, when a semaphore becomes free it can send events to a task that has requested to be notified of the semaphore's change of state. This functionality is not described in this library. Please refer to the documentation for semEvLib and msgQEvLib.</p>
EXAMPLE	<p>An ISR defers device error handling to a worker task that is capable of identifying which device suffered the error based on the event number it received from the ISR. This example assumes the ISR is already connected to the proper interrupt vector and the workerTask has already been spawned. It is also assumed that various initialization steps have already been taken such as variables and list initializations.</p>

```
#include <vxWorks.h>
#include <device.h>      /* Fictitious library for DEVICE related
definitions */
#include <dllLib.h>
```

```

#include <eventLib.h>

/* externs */

extern int ffsLsb (UINT32 i); /* find first set bit in 32 bit-word */

/* forward declarations */

DEVICE * devEventNumToDevPtr (UINT32 * pEvent);

/* globals */

int workerTaskId;          /* Worker task ID.  Initialized before */
                          /* devIntHandler ever runs */

DL_LIST devList; /* Device list.  Initialized before */
                /* the workerTask ever runs */

void devIntHandler          /* Device interrupt handler */
(
    DEVICE * pDevice        /* device that needs servicing */
)
{
    if ((pDevice->state & DEVICE_ERROR_MASK) == 0)
    {
        /* Device not in error state.  Process interrupt now */

    }
    else
    {
        /*
         * Defer processing of error to task by sending events to
         * the worker task.  It is assumed the device was assigned
         * a unique power-of-two eventNumber at creation so the
         * worker task can identify it amongst a number of similar
         * devices present in the system.
         *
         * Passing pDevice as an event directly is not a good
         * solution because this interrupt handler could run
         * again before the worker task has a chance to read its
         * events register.
         */

        eventSend (workerTaskId, pDevice->eventNumber);

        /*
         * DO NOT re-enable the device until the worker task has
         * dealt with the error condition.
         */
    }
}

void workerTask ()
{

```

```
    UINT32 events;          /* where event are copied */
    STATUS result;          /* eventReceive() call result */
    DEVICE * pDevice;       /* failed device */

    while (TRUE)
    {

/*
 * Wait for any of 32 events. It is possible that more than
 * one event be received if two or more devices are found
 * to have errors before this task gets the CPU.
 */
        result = eventReceive (0xffffffff, EVENTS_WAIT_ANY,
                                WAIT_FOREVER, &events);

        if (result != OK)
        {
            /*
             * Failed to receive events. Perform some sort of
             * error handling
             */
        }

/* Process every event in events */
        while ((pDevice = devEventNumToDevPtr (&events)) != NULL)
        {
            /* Process error on pDevice. Perhaps re-enable the device */
        }

        if (events != 0)
        {
            /* An error has occurred since events should be 0 at
             * this point. Somehow devEventNumToDevPtr() could
             * not map a device to the last value of event it
             * received.
             */
        }
    }
}

DEVICE * devEventNumToDevPtr
(
    UINT32 * pEvents        /* events to process */
)
{
    {
        UINT32 eventNumber;    /* stores one and one event only */
        DEVICE * pDevice = NULL; /* returned value */

        /* Find an event in pEvents */

        if ((eventNumber = ffsLsb (*pEvents)) != 0)
        {
            eventNumber = 1 << (eventNumber - 1);
        }

        /* Find the device which maps to eventNumber */
    }
}
```

```

        if (eventNumber != 0)
        {
            devListLock(); /* Lock the device list. Fictitious rtn */

            /* Get the first device in the list */

            pDevice = (DEVICE *) DLL_FIRST (&devList);

            while (pDevice != NULL)
            {
                if (pDevice->eventNumber == eventNumber)
                {
                    /*
                     * Found the device in the list. Stop searching
                     * and clear the event number so next time this
                     * routine is called the next event in pEvent will
                     * be processed.
                     */
                    *pEvents = *pEvents & ~(1 << (eventNumber - 1))
                    break;
                }

                pDevice = (DEVICE *) DLL_NEXT (pDevice);
            }
            devListUnlock(); /* Unlock the device list. Fictitious rtn */
        }

        return (pDevice);
    }

```

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **eventLib.h**

SEE ALSO **eventShow**, **semEvLib**, **msgQEvLib**, the VxWorks programmer's guides

excArchLib

NAME	excArchLib – architecture-specific exception-handling facilities
ROUTINES	excVecInit() – initialize the exception/interrupt vectors excConnect() – connect a C routine to an exception vector (PowerPC) excIntConnect() – connect a C routine to an asynchronous exception vector (PowerPC, ARM) excCrtConnect() – connect a C routine to a critical exception vector (PowerPC 403) excIntCrtConnect() – connect a C routine to a critical interrupt vector (PowerPC 403) excVecSet() – set a CPU exception vector (PowerPC, ARM) excVecGet() – get a CPU exception vector (PowerPC, ARM)
DESCRIPTION	This library contains exception-handling facilities that are architecture dependent. For information about generic (architecture-independent) exception-handling, see the manual entry for excLib .
INCLUDE FILES	excLib.h
SEE ALSO	excLib, dbgLib, sigLib, intLib

excLib

NAME	excLib – generic exception handling facilities
ROUTINES	excInit() – initialize the exception handling package excJobAdd() – request a task-level function call from interrupt level excHookAdd() – specify a routine to be called with exceptions
CONFIGURATION	To use this functionality, configure the INCLUDE_EXC_TASK component. This component also takes a configuration parameter MAX_ISR_JOBS which must always be a power of 2, and no larger than 64K.
ADDITIONAL EXCEPTION HANDLING ROUTINE	The excHookAdd() routine adds a routine that will be called when a hardware exception occurs. This routine is called at the end of normal exception handling.
DBGLIB	The facilities of excLib , including excTask() , are used by dbgLib to support breakpoints, single-stepping, and additional exception handling functions.

SIGLIB	A higher-level, UNIX-compatible interface for hardware and software exceptions is provided by sigLib . If sigvec() is used to initialize the appropriate hardware exception/interrupt (e.g., BUS ERROR == SIGSEGV), excLib will use the signal mechanism instead.
INCLUDE FILES	excLib.h
SEE ALSO	dbgLib , sigLib , intLib

fccVxbEnd

NAME	fccVxbEnd – fcc VxBus END driver
ROUTINES	fccRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver for the Motorola/Freescale Fast Communications Controller (FCC) Ethernet network interface. The FCC supports several communication protocols. This driver supports the FCC operating in Ethernet mode, which is fully compliant with the IEEE 802.3u 10Base-T and 100Base-T specifications. When programmed for ethernet mode, the FCC is used in conjunction with an external MII-compliant PHY.</p> <p>The FCC exists in various Freescale CPUs that contain a Communications Processor Module (CPM), including the MPC8260, MPC8272, and MPC8560. There may be up to 3 FCCs, each of which may be programmed for different modes, including ethernet, HDLC and transparent mode. Using the FCC for ethernet requires access to 5 different resources within the processor:</p> <ul style="list-style-type: none"> - A region of dual port RAM containing various configuration registers (PRAM) - A region of internal memory mapped registers (IRAM) - The CP command register (CPCR) - Various parallel I/O registers (used for RX/TX signals and MDIO/PHY access) - A single interrupt vector <p>This VxBus driver manages access only to the PRAM and IRAM register regions and the interrupt vector. Access to the CP command register is managed through a separate VxBus cpm driver, and access to the PHY is provided through a separate VxBus mdio driver. (The PHY on the mdio driver's MII bus is remapped to the FCC port via the BSP's hwconf.c file.)</p>
BOARD LAYOUT	The FCC is directly integrated into the CPU. All configurations are jumperless.
EXTERNAL INTERFACE	<p>The driver provides a vxBus external interface. The only exported routine is the fccRegister() function, which registers the driver with VxBus.</p>

In order to enable the FCC, several of the parallel I/O pins must be programmed to connect the FCC's RX and TX signals to the external PHY, and to supply the necessary clock signals. This setup is board-specific, and is not handled directly by this driver. Instead, the driver will query its parent bus (which will always be the PLB, since the FCC is a processor local bus device) for enable and disable methods. These methods should be registered with the PLB by the BSP during VxBus initialization. If an enable method is found, it will be called by the **fccInstInit()** routine. If a disable method is found, it will be called during the **fccInstUnlink()** routine.

INCLUDE FILES none

SEE ALSO vxBus, **ifLib**, "Writing an Enhanced Network Driver", "MPC8260 PowerQUICC II Family Reference Manual", <http://www.freescale.com/files/product/doc/MPC8260UM.pdf>

fecVxbEnd

NAME **fecVxbEnd** – Motorola/Freescale FEC VxBus END driver

ROUTINES **fecRegister()** – register with the VxBus subsystem

DESCRIPTION This module implements a driver for the Motorola/Freescale Fast Ethernet Controller (FEC) found on PowerPC 8xx and Coldfire embedded processors. The FEC is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications and supports both 7 wire serial mode (10Mbps only) and MII transceivers (10/100 Mbps in full or half duplex).

The FEC is a processor local bus device. Each controller has a bank of control/status registers (CSR) which is mapped into host address space. Packet transfers are performed through DMA using separate receive and transmit DMA rings. Each ring is constructed from descriptors that are 8 bytes in size and can address a single data buffer at any 32-bit address within host RAM. Multiple descriptors can be chained together to perform scatter/gather DMA.

The PowerPC 8xx version of the FEC is a bus master: all DMA operations are performed through the FEC itself. The Coldfire FEC requires the use of the Multi-Channel DMA controller, which is accessed through the VxBus slave DMA API. (A VxBus slave DMA driver is provided.)

BOARD LAYOUT The FEC is directly integrated into the CPU. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides the standard VxBus external interface, **fecRegister()**. This function registers the driver with the VxBus subsystem, and instances will be created as needed.

Since the FEC is a processor local bus device, each device instance must be specified in the **hwconf.c** file in a BSP. The hwconf entry must specify the following parameters:

regBase

Specifies the base address where the controller's CSR registers are mapped into the host's address space. All register offsets are computed relative to this address. For the PPC 8xx architecture, the base address is usually MBAR + 0xE00. For the Coldfire FEC, the base address can be MBAR + 0x9000 and (for Coldfire boards with two FEC ports) MBAR + 0x9800.

intr

Specifies the interrupt vector for the FEC. For the PPC FEC, this designates which bit in the SIPEND register in the SIU will be assigned to the FEC.

intrLevel

Specifies the interrupt level for the FEC. Currently, this is the same value as interrupt vector.

phyAddr

The MII management address (0-31) of the PHY for this particular FEC device. Each FEC typically has at least one PHY allocated to it (unless it's in serial mode).

miiIfName

The name of the driver that provides the MII interface for this FEC unit. On boards that have multiple FEC devices (e.g. the ads885), the management pins for all of the PHYs will all be wired to the MDIO pins on just one controller. The *miiIfName* resource (and the *miiIfUnit* resource below) are used to tell each FEC instance which one is the management controller. If a device is not the management controller, it will just forward its PHY register read and write requests to the one that is.

miiIfUnit

The unit number of the device that provides the MII management methods for this FEC instance.

An example hwconf entry is shown below:

```
const struct hcfResource motFecHEnd0Resources[] = {
    { "regBase", HCF_RES_INT, { (void *) (MBAR_VAL + 0x9000) } },
    { "intr", HCF_RES_INT, { (void *) 67 } },
    { "intrLevel", HCF_RES_INT, { (void *) 67 } },
    { "phyAddr", HCF_RES_INT, { (void *) 0 } },
    { "miiIfName", HCF_RES_STRING, { (void *) "motfec" } },
    { "miiIfUnit", HCF_RES_INT, { (void *) 0 } }
};
```

The FEC driver uses the ifmedia interface, which allows media selection to be controller by the ifconfig utility, or the SIOCGIFMEDIA/SIOCSIFMEDIA ioctl API. It also uses the miiBus subsystem to manage its PHYs, so no MII handling code is needed within the FEC driver itself.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function on the PPC and the Coldfire architectures:

sysFecEnetAddrGet ()

STATUS sysFecEnetAddrGet (UINT32, UINT8 *)

This function is used on both the PPC and Coldfire FEC, and is the only one required for the PPC FEC. This routine queries the BSP to provide the ethernet address for a given FEC unit. (For PPC, the unit number is always 0, since the PPC 8xx devices only have one FEC port.)

INCLUDE FILES none

SEE ALSO vxBus, miiBus, **ifLib**, **endLib**, "Writing an Enhanced Network Driver", "MPC885 PowerQUICC Family Reference Manual
http://www.freescale.com/files/32bit/doc/ref_manual/MPC885RM.pdf", "MCF5475 Reference Manual http://www.freescale.com/files/32bit/doc/ref_manual/MCF5475RM.pdf"

fei8255xVxbEnd

NAME fei8255xVxbEnd – Intel PRO/100 VxBus END driver

ROUTINES feiRegister () – register with the VxBus subsystem

DESCRIPTION This module provides driver support for the Intel PRO/100 series of 10/100 ethernet controllers. This includes the i82557, i82558, i82559, i82550, i82551, and the embedded PRO/100 interfaces in several Intel motherboard chipsets. The 8255x family is compatible with the IEEE 802.3 10Base-T and 100base-T specifications. It features a glueless 32-bit PCI bus master interface that complies with the PCI v2.1 specification. An interface to MII compliant physical layer devices is built-in to the controller. The 8255x also includes Flash support up to 1 MByte and EEPROM support. The flash module is not used in this driver.

The 8255x has two DMA channels. One of them, the RX DMA channel, is used exclusively for reception of ethernet frames. The other is the command DMA channel, which is used by the host to execute commands on the controller. One of these commands happens to be the 'transmit ethernet frame' command, however it's a mistake to think of the command DMA channel as the TX DMA ring only. The same channel is used to issue configuration commands, to set the station address, and to program the multicast filter. This aspect of the controller makes the driver design a little tricky, since access to the command DMA channel is effectively shared between the driver's data and control planes.

For frame reception and transmission, the 8255x offers two types of DMA descriptor schemes, referred to as simple mode and flexible mode. In simple mode, a frame must be

contained in a single contiguous buffer, both for reception and transmission. With flexible transmit mode, each TX descriptor uses a fragment array that allows for in-place scatter/gather DMA of an arbitrary number of fragments. Flexible receive mode is not quite as straightforward to use as flexible transmit mode, however it has the advantage of allowing RX buffers to be placed at arbitrarily aligned locations in memory (by contrast, simple RX mode imposes certain alignment restrictions).

In simple RX mode, each RX descriptor has a packet data region immediately following it, into which the chip will deposit incoming frames. In effect, the RX descriptor is treated as a header attached to the packet. This makes managing the RFDs tricky since the entire RFD must be removed from the RX DMA ring in order to loan the packet out to the stack for processing.

In flexible RX mode, the RFD has no frame data area. Instead, the RFD ring is accompanied by a second ring of descriptors known as RBDs. The first RBD pointer in the first RFD in the ring is used to tell the chip the location of the start of the RBD list. It's possible to have a different number of RBDs than RFDs. Each entry in the RBD list is basically just a data buffer that the chip will fill in with received packet data. If the RBD happens to be large enough to hold a full size ethernet frame, only one RBD per packet will be needed.

To keep things simple, the driver uses the convention that each RFD has one accompanying RBD associated with it. The RBD buffers are all large enough to contain a full size frame, so the chip should always consume one RFD and one RBD per packet. The RFD and RBD structures together are treated as a single descriptor structure by the driver, even though technically the chip treats them as separate entities.

The 82550 and 82551 offer additional features including TCP/IP checksum offload and VLAN tag stripping and insertion. Using these features involves the use of extended RX and TX descriptor formats, which must be enabled via special bits in the configuration block.

The 82550 appears to have an erratum that causes it to miscalculate IP header checksums for certain very small datagrams, particularly IP fragments from 1 to 3 bytes in size. These may occur if you transmit an IP datagram that's slightly larger than your MTU size. It's possible to work around this problem by checking for IP packets of this size and calculating the checksum in software, however this approach has a couple of drawbacks: it introduces a linker dependency on the `in_cksum()` routine in the TCP/IP stack, and it's possible that the extra processing needed to test for the error case might negate the benefit of the IP header checksum offload in the first place. By default, the driver is compiled such that it will not enable transmit IP header checksum support for the 82550 device. If the driver is compiled with the `FEI_IP_HDR_CSUM_WAR` macro defined, the software workaround will be enabled. The workaround is unnecessary for the 82551 chip, in which the erratum appears to have been corrected.

The 82557, 82558 and 82559 do not support the special checksum offload and VLAN features. For those chips, this driver will use the flexible TX and RX DMA mode, but checksum offload and VLAN stripping/insertion are disabled. For the 82550 and 82551, the driver will use the extended TxCB and extended RFD formats instead. The chip type will be automatically detected and the features enabled without any intervention from the user.

The checksum offload and VLAN features can be disabled at runtime using the `ENDIFCAP ioctls`, if desired.

BOARD LAYOUT The 8255x controllers are available in standalone PCI and cardbus adapter formats, and may also be directly integrated onto the system main board. There are also Intel x86 motherboard chipsets with built-in PRO/100 ethernet devices. All configurations are jumperless.

EXTERNAL INTERFACE The driver provides a VxBus external interface. The only exported routine is the **feiRegister()** function, which registers the driver with VxBus.

INCLUDE FILES **fei8255xVxbEnd.h** **end.h** **endLib.h** **netBufLib.h** **muxLib.h**

SEE ALSO **vxBusLib**, **ifLib**, **endLib**, *"Writing an Enhanced Network Driver"*, *"Intel(r) 8255x 10/100 Mbps Ethernet Controller Family Open Source Software Developer Manual"*, http://developer.intel.com/design/network/manuals/8255x_opensdm.htm

ffsLib

NAME **ffsLib** – find first bit set library

ROUTINES **ffsMsb()** – find most significant bit set
 ffsLsb() – find least significant bit set

DESCRIPTION This library contains routines to find the first bit set in a 32 bit field. It is utilized by bit mapped priority queues and hashing functions.

INCLUDE FILES **ffsLib.h**

fioBaseLib

NAME **fioBaseLib** – formatted I/O library

ROUTINES **fioBaseLibInit()** – initialize the formatted I/O support library
 printf() – write a formatted string to the standard output stream (ANSI)
 oprintf() – write a formatted string to an output function
 printErr() – write a formatted string to the standard error stream
 sprintf() – write a formatted string to a buffer (ANSI)

snprintf() – write a formatted string to a buffer, not exceeding buffer size (ANSI)
fioFormatV() – convert a format string

DESCRIPTION

This library provides the basic formatting and scanning I/O functions. It includes some routines from the ANSI-compliant **printf()**/**scanf()** family of routines. It also includes several utility routines.

If the floating-point format specifications **e**, **E**, **f**, **g**, and **G** are to be used with these routines, the routine **floatInit()** must be called first. If the configuration macro **INCLUDE_FLOATING_POINT** is defined, **floatInit()** is called by the root task, **usrRoot()**, in **usrConfig.c**.

These routines do not use the buffered I/O facilities provided by the standard I/O facility. Thus, they can be invoked even if the standard I/O package has not been included. This includes **printf()**, which in most UNIX systems is part of the buffered standard I/O facilities. Because **printf()** is so commonly used, it has been implemented as an unbuffered I/O function. This allows minimal formatted I/O to be achieved without the overhead of the entire standard I/O package. For more information, see the manual entry for **ansiStdio**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and **intCpuLock** restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are **intCpuLock** restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES

fioLib.h, **stdio.h**

SEE ALSO

ansiStdio, **floatLib**, *"VxWorks Kernel Programmer's Guide: I/O System"*

fioLib

NAME

fioLib – formatted I/O library

ROUTINES

fioLibInit() – initialize the formatted I/O support library
voprintf() – write a formatted string to an output function
fdprintf() – write a formatted string to a file descriptor
vprintf() – write a string formatted with a variable argument list to standard output (ANSI)
vfdprintf() – write a string formatted with a variable argument list to a file descriptor
vsprintf() – write a string formatted with a variable argument list to a buffer (ANSI)

vsprintf() – write a string formatted with a variable argument list to a buffer, not exceeding buffer size (ANSI)

fioRead() – read a buffer

fioRdString() – read a string from a file

sscanf() – read and convert characters from an ASCII string (ANSI)

DESCRIPTION

This library provides the basic formatting and scanning I/O functions. It includes some routines from the ANSI-compliant **printf()**/**scanf()** family of routines. It also includes several utility routines.

If the floating-point format specifications **e**, **E**, **f**, **g**, and **G** are to be used with these routines, the routine **floatInit()** must be called first. If the configuration macro **INCLUDE_FLOATING_POINT** is defined, **floatInit()** is called by the root task, **usrRoot()**, in **usrConfig.c** for BSP builds or in **prjConfig.c** in project builds.

These routines do not use the buffered I/O facilities provided by the standard I/O facility. Thus, they can be invoked even if the standard I/O package has not been included. This includes **printf()**, which in most UNIX systems is part of the buffered standard I/O facilities. Because **printf()** is so commonly used, it has been implemented as an unbuffered I/O function. This allows minimal formatted I/O to be achieved without the overhead of the entire standard I/O package. For more information, see the manual entry for **ansiStdio**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and **intCpuLock** restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are **intCpuLock** restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES

fioLib.h, **stdio.h**

SEE ALSO

ansiStdio, **floatLib**, *"VxWorks Programmer's Guide: I/O System"*

fppArchLib

NAME

fppArchLib – architecture-dependent floating-point coprocessor support

ROUTINES

fppSave() – save the floating-point coprocessor context

fppRestore() – restore the floating-point coprocessor context

fppProbe() – probe for the presence of a floating-point coprocessor

fppTaskRegsGet() – get the floating-point registers from a task TCB

fppTaskRegsSet() – set the floating-point registers of a task

DESCRIPTION This library contains architecture-dependent routines to support the floating-point coprocessor. The routines **fppSave()** and **fppRestore()** save and restore all the task floating-point context information. The routine **fppProbe()** checks for the presence of the floating-point coprocessor. The routines **fppTaskRegsSet()** and **fppTaskRegsGet()** inspect and set coprocessor registers on a per-task basis.

With the exception of **fppProbe()**, the higher-level facilities in **dbgLib** and **usrLib** should be used instead of these routines. For information about architecture-independent access mechanisms, see the manual entry for **fppLib**.

INITIALIZATION To activate floating-point support, **fppInit()** must be called before any tasks using the coprocessor are spawned. This is done by the root task, **usrRoot()**, in **usrConfig.c**. See the manual entry for **fppLib**.

NOTE X86 There are two kind of floating-point contexts and set of routines for each kind. One is 108 bytes for older FPU (i80387, i80487, Pentium) and older MMX technology and **fppSave()**, **fppRestore()**, **fppRegsToCtx()**, and **fppCtxToRegs()** are used to save and restore the context, convert to or from the **FPPREG_SET**. The other is 512 bytes for newer FPU, newer MMX technology and streaming SIMD technology (PentiumII, III, 4) and **fppXsave()**, **fppXrestore()**, **fppXregsToCtx()**, and **fppXctxToRegs()** are used to save and restore the context, convert to or from the **FPPREG_SET**. Which to use is automatically detected by checking CPUID information in **fppArchInit()**. And **fppTaskRegsSet()** and **fppTaskRegsGet()** access the appropriate floating-point context. The bit interrogated for the automatic detection is the "Fast Save and Restore" feature flag.

NOTE X86 INITIALIZATION

To activate floating-point support, **fppInit()** must be called before any tasks using the coprocessor are spawned. If **INCLUDE_FLOATING_POINT** is defined in **configAll.h**, this is done by the root task, **usrRoot()**, in **usrConfig.c**.

NOTE X86 VX_FP TASK OPTION

Saving and restoring floating-point registers adds to the context switch time of a task. Therefore, floating-point registers are *not* saved and restored for *every* task. Only those tasks spawned with the task option **VX_FP_TASK** will have floating-point state, MMX technology state, and streaming SIMD state saved and restored.

NOTE: If a task does any floating-point operations, MMX operations, and streaming SIMD operation, it must be spawned with **VX_FP_TASK**. It is deadly to execute any floating-point operations in a task spawned without **VX_FP_TASK** option, and very difficult to find. To detect that illegal/unintentional/accidental floating-point operations, a new API and mechanism is added. The mechanism is to enable or disable the FPU by toggling the TS flag in the CR0 in the new task switch hook routine - **fppArchSwitchHook()** - respecting the **VX_FP_TASK** option. If **VX_FP_TASK** option is not set in the switching-in task, the FPU is disabled. Thus the device-not-available exception will be raised if that task does any

floating-point operations. This mechanism is disabled in the default. To enable, call the enabler - **fppArchSwitchHookEnable()** - with a parameter **TRUE(1)**. A parameter **FALSE(0)** disables the mechanism.

NOTE X86 MIXING MMX AND FPU INSTRUCTIONS

A task with **VX_FP_TASK** option saves and restores the FPU and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FPU and MMX state if the FPU and MMX instructions are not mixed within a task. Because the MMX registers are aliased to the FPU registers, care must be taken when making transitions between FPU instructions and MMX instructions to prevent the loss of data in the FPU and MMX registers and to prevent incoherent or unexpected result. When mixing MMX and FPU instructions within a task, follow these guidelines from Intel:

- Keep the code in separate modules, procedures, or routines.
- Do not rely on register contents across transitions between FPU and MMX code modules.
- When transitioning between MMX code and FPU code, save the MMX register state (if it will be needed in the future) and execute an EMMS instruction to empty the MMX state.
- When transitioning between FPU and MMX code, save the FPU state, if it will be needed in the future.

NOTE X86 MIXING SSE SSE2 FPU AND MMX INSTRUCTIONS

The XMM registers and the FPU/MMX registers represent separate execution environments, which has certain ramifications when executing SSE, SSE2, MMX and FPU instructions in the same task context:

- Those SSE and SSE2 instruction that operate only on the XMM registers (such as the packed and scalar floating-point instructions and the 128-bit SIMD integer instructions) can be executed in the same instruction stream with 64-bit SIMD integer or FPU instructions without any restrictions. For example, an application can perform the majority of its floating-point computations in the XMM registers, using the packed and scalar floating-point instructions, and at the same time use the FPU to perform trigonometric and other transcendental computations. Likewise, an application can perform packed 64-bit and 128-bit SIMD integer operations can be executed together without restrictions.
- Those SSE and SSE2 instructions that operate on MMX registers (such as the CVTTPS2PI, CVTTPS2PI, CVTPI2PS, CVTPD2PI, CVTTPD2PI, CVTPI2PD, MOVDQ2Q, MOVQ2DQ, PADDQ, and PSUBQ instructions) can also be executed in the same instruction stream as 64-bit SIMD integer or FPU instructions, however, here they subject to the restrictions on the simultaneous use of MMX and FPU instructions, which mentioned in the previous paragraph.

NOTE X86 INTERRUPT LEVEL

Floating-point registers are *not* saved and restored for interrupt service routines connected with **intConnect()**. However, if necessary, an interrupt service routine can save and restore floating-point registers by calling routines in **fppALib**. See the manual entry for **intConnect()** for more information.

NOTE X86 EXCEPTIONS

There are six FPU exceptions that can send an exception to the CPU. They are controlled by Exception Mask bits of the Control Word register. VxWorks disables them in the default configuration. They are:

- Precision
- Overflow
- Underflow
- Division by zero
- Denormalized operand
- Invalid Operation

The FPU in 486 or later IA32 processors provide two different modes to handle a FPU floating-point exceptions. MSDOS compatibility mode and native mode. The mode of operation is selected with the NE flag of control register CR0. The MSDOS compatibility mode is not supported, because it is old and requires external signal handling. The native mode for handling FPU exceptions is used by setting the NE flag in the control register CR0. In this mode, if the FPU detects an exception condition while executing a floating-point instruction and the exception is unmasked (the mask bit for the exception is cleared), the FPU sets the flag for the exception and the ES flag in the FPU status word. It then invokes the software exception handler through the floating-point-error exception (vector number 16), immediately before execution of any of the following instructions in the processor's instruction stream:

- The next floating-point instruction, unless it is one of the non-waiting instructions (FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV and FNSAVE).
- The next WAIT/FWAIT instruction.
- The next MMX instruction.

If the next floating-point instruction in the instruction stream is a non-waiting instruction, the FPU executes the instruction without invoking the software exception handler. There is a well known FPU exception synchronization problems that occur in the time frame between the moment when the exception is signaled and when it is actually handled. Because of concurrent execution (integer unit and FPU), integer or system instructions can be executed during this time frame. It is thus possible for the source or destination operands for a floating-point instruction that faulted to be overwritten in memory, making it impossible for the exception handler to analyze or recover from the exception. To solve this problem, an exception synchronizing instruction (either a floating-point instruction or a WAIT/FWAIT instruction) can be placed immediately after any floating-point instruction that might present a situation where state information pertaining to a floating-point

exception might be lost or corrupted. The preemption could happen at any instruction boundary that maybe right after the faulting instruction, and could result in the task context switch. The task context switch does not perform the FPU context switch always for optimization, and the FPU context switch maybe done in other task context. To make the pending unmasked exceptions to be handled in the task context that it happened, the FPU context switch does not check pending unmasked exceptions, and preserves the exception flags in the status register. It may not be useful to re-execute the faulting instruction, if the faulting floating-point instruction is followed by one or more non-floating-point instructions. The return instruction pointer on the stack (exception stack frame) may not point to the faulting instruction. The faulting instruction pointer is contained in the saved FPU state information. The default exception handler does not replace the return instruction pointer with the faulting instruction pointer. **fppCwSet()** and **fppCwGet()**, sets and gets the X86 FPU control word. **fppSwGet()** gets the X86 FPU status word. **fppWait()** checks for pending unmasked FPU exceptions. **fppClex()** clears FPU exception flags after checking unmasked FPU pending exceptions. **fppNclex()** clears FPU exception flags without checking unmasked FPU pending exceptions.

NOTE ARM This architecture does not currently support floating-point coprocessors.

INCLUDE FILES **fppLib.h**

SEE ALSO **fppLib**, **intConnect()**, *"Motorola MC68881/882 Floating-Point Coprocessor User's Manual"*, *"Intel 387 DX User's Manual"*, *"Intel Architecture Software Developer's Manual"*, *"Renesas SH7750 Hardware Manual"*, Gerry Kane and Joe Heinrich, *"MIPS RISC Architecture Manual"*

fppLib

NAME **fppLib** – floating-point coprocessor support library

ROUTINES **fppInit()** – initialize floating-point coprocessor support

DESCRIPTION This library provides a general interface to the floating-point coprocessor. To activate floating-point support, **fppInit()** must be called before any tasks using the coprocessor are spawned. This is done automatically by the root task, **usrRoot()**, in **usrConfig.c** when the configuration macro **INCLUDE_HW_FP** is defined.

For information about architecture-dependent floating-point routines, see the manual entry for **fppArchLib**.

The **fppShow()** routine displays coprocessor registers on a per-task basis. For information on this facility, see the manual entries for **fppShow** and **fppShow()**.

VX_FP_TASK OPTION Saving and restoring floating-point registers adds to the context switch time of a task. Therefore, floating-point registers are not saved and restored for every task. Only those tasks spawned with the task option **VX_FP_TASK** will have floating-point registers saved and restored.

NOTE If a task does any floating-point operations, it must be spawned with **VX_FP_TASK**.

INTERRUPT LEVEL

Floating-point registers are not saved and restored for interrupt service routines connected with **intConnect()**. However, if necessary, an interrupt service routine can save and restore floating-point registers by calling routines in **fppArchLib**.

INCLUDE FILES **fppLib.h**

SEE ALSO **fppArchLib**, **fppShow**, **intConnect()**, *"VxWorks Kernel Programmer's Guide: Basic OS"*

fppShow

NAME **fppShow** – floating-point show routines

ROUTINES **fppShowInit()** – initialize the floating-point show facility
fppTaskRegsShow() – print the contents of a task's floating-point registers

DESCRIPTION This library provides the routines necessary to show a task's optional floating-point context. To use this facility, it must first be installed using **fppShowInit()**, which is called automatically when the floating-point show facility is configured into VxWorks using either of the following methods:

- If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.
- If you use the Tornado project facility, select **INCLUDE_HW_FP_SHOW**.

This library enhances task information routines, such as **ti()**, to display the floating-point context.

INCLUDE FILES **fppLib.h**

SEE ALSO **fppLib**

fsEventUtilLib

NAME	fsEventUtilLib – Event Utility functions for different file systems
ROUTINES	fsEventUtilInit() – Initialize the file system event utility library fsPathAddedEventSetup() – Setup to wait for a path fsPathAddedEventRaise() – Raise a "path added" event fsWaitForPath() – wait for a path
DESCRIPTION	This library contains file systems event utility routines.
INCLUDE FILES	fsEventUtilLib.h

fsMonitor

NAME	fsMonitor – The File System Monitor
ROUTINES	fsMonitorInit() – Initialize the fsMonitor fsmNameMap() – map an XBD name to a Core I/O path fsmProbeInstall() – install F/S probe and instantiator functions fsmProbeUninstall() – remove a file system probe fsmNameInstall() – Add a mapping between an XBD name and a pathname fsmNameUninstall() – remove an XBD name to pathname mapping fsmGetDriver() – Get the XBD name of a mapping based on the path fsmGetVolume() – get the pathname based on an XBD name mapping fsmUnmountHookAdd() – Add an unmount hook function fsmUnmountHookDelete() – Remove an unmount hook function fsmUnmountHookRun() – Runs the unmount hook functions
DESCRIPTION	This library implements the File System Monitor, which controls the autodetection and instantiation of File Systems.
INCLUDE FILES	fsMonitor.h

fsPxLib

NAME	fsPxLib – I/O, file system API library (POSIX)
-------------	---

ROUTINES	unlink() – unlink a file link() – link a file fsync() – synchronize a file fdatasync() – synchronize a file data rename() – change the name of a file fpathconf() – determine the current value of a configurable limit pathconf() – determine the current value of a configurable limit access() – determine accessibility of a file chmod() – change the permission mode of a file fchmod() – change the permission mode of a file
DESCRIPTION	This library contains POSIX APIs which are applicable to I/O, file system.
INCLUDE FILES	ioLib.h , stdio.h
SEE ALSO	ioLib , iosLib , <i>"VxWorks Kernel Programmer's Guide: I/O System"</i>

ftruncate

NAME	ftruncate – POSIX file truncation
ROUTINES	ftruncate() – truncate a file (POSIX)
DESCRIPTION	This module contains the POSIX compliant ftruncate() routine for truncating a file.
INCLUDE FILES	unistd.h
SEE ALSO	

gei825xxVxbEnd

NAME	gei825xxVxbEnd – Intel PRO/1000 VxBus END driver
ROUTINES	geiRegister() – register with the VxBus subsystem
DESCRIPTION	This module implements a driver for the Intel PRO/1000 series of gigabit ethernet controllers. The PRO/1000 family includes PCI, PCI-X, PCIe and CSA adapters.

The PRO/1000 controllers implement all IEEE 802.3 receive and transmit MAC functions. They provide a Ten-Bit Interface (TBI) as specified in the IEEE 802.3z standard for 1000Mb/s full-duplex operation with 1.25 GHz Ethernet transceivers (SERDES), as well as a GMII interface as specified in IEEE 802.3ab for 10/100/1000 BASE-T transceivers, and also an MII interface as specified in IEEE 802.3u for 10/100 BASE-T transceivers.

Enhanced features available in the PRO/1000 family include TCP/IP checksum offload for both IPv4 and IPv6, VLAN tag insertion and stripping, VLAN tag filtering, TCP segmentation offload, interrupt coalescing, hardware RMON statistics counters, 64-bit addressing and jumbo frames. This driver makes use of the checksum offload, VLAN tag insertion/stripping and jumbo frames features, as available.

Note that not all features are available on all devices. The 82543 does not support IPv4 RX checksum offload due to a hardware bug. IPv6 checksum offload is available on transmit for all adapters, but only available on receive with adapters newer than the 82544.

Currently, this driver supports the 82543, 82544, 82540, 82541, 82545, 82546, 82571, 82572, and 82573 controllers with copper UTP and TBI multimode fiber media only (SERDES adapters have not been tested).

BOARD LAYOUT The PRO/1000 is available on standalone PCI, PCI-X and PCIe NICs as well as integrated onto various system boards. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **geiRegister()** function, which registers the driver with VxBus.

The PRO/1000 devices also support jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of a buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface gei0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "gei", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

INCLUDE FILES **gei825xxVxbEnd.h** **geiTbiPhy.h** **end.h** **endLib.h** **netBufLib.h** **muxLib.h**

SEE ALSO vxBus, **ifLib**, **endLib**, **netBufLib**, miiBus, "Intel PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual"
http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf", "Intel PCIe GbE Controllers Open Source Software Developer's Manual"
<http://download.intel.com/design/network/manuals/31608001.pdf>", "Intel 82544EI/82544GC Gigabit Ethernet Controller Specification Update"
http://download.intel.com/design/network/specupdt/82544_a4.pdf", "Intel 82540EM Gigabit Ethernet Controller Specification Update"
http://download.intel.com/design/network/specupdt/82540em_a2.pdf", "Intel 82545EM Gigabit Ethernet Controller Specification Update"

<http://download.intel.com/design/network/specupdt/82545em.pdf>", "Intel 82573 Family Gigabit Ethernet Controllers Specification Update and Sighting Information"
<http://download.intel.com/design/network/specupdt/82573.pdf>"

getopt

NAME	getopt – getopt facility
ROUTINES	getopt() – parse argc/argv argument vector (POSIX) getoptInit() – initialize the getopt state structure getopt_r() – parse argc/argv argument vector (POSIX) getOptServ() – parse parameter string into argc, argv format
DESCRIPTION	<p>This library supplies both a POSIX compliant getopt() which is a command line parser, as well as a reentrant version of the same command named getopt_r(). Prior to calling getopt_r(), the caller needs to initialize the getopt state structure by calling getoptInit(). This explicit initialization is not needed while calling getopt() as the system is setup as if the initialization has already been done.</p> <p>The user can modify getopt() behavior by setting the the getopt variables like optind, opterr, etc. For getopt_r(), the value needs to be updated in the getopt state structure.</p>
INCLUDE FILES	none

hashLib

NAME	hashLib – generic hashing library
ROUTINES	hashTblCreate() – create a hash table hashTblInit() – initialize a hash table hashTblDelete() – delete a hash table hashTblTerminate() – terminate a hash table hashTblDestroy() – destroy a hash table hashTblPut() – put a hash node into the specified hash table hashTblFind() – find a hash node that matches the specified key hashTblRemove() – remove a hash node from a hash table hashTblEach() – call a routine for each node in a hash table hashFuncIterScale() – iterative scaling hashing function for strings hashFuncModulo() – hashing function using remainder technique

hashFuncMultiply() – multiplicative hashing function
hashKeyCmp() – compare keys as 32 bit identifiers
hashKeyStrCmp() – compare keys based on strings they point to

DESCRIPTION	This subroutine library supports the creation and maintenance of a chained hash table. Hash tables efficiently store hash nodes for fast access. They are frequently used for symbol tables, or other name to identifier functions. A chained hash table is an array of singly linked list heads, with one list head per element of the hash table. During creation, a hash table is passed two user-definable functions, the hashing function, and the hash node comparator.
CONFIGURATION	To use the generic hashing library, configure VxWorks with the INCLUDE_HASH component.
HASH NODES	A hash node is a structure used for chaining nodes together in the table. The defined structure HASH_NODE is not complete because it contains no field for the key for referencing, and no place to store data. The user completes the hash node by including a HASH_NODE in a structure containing the necessary key and data fields. This flexibility allows hash tables to better suit varying data representations of the key and data fields. The hashing function and the hash node comparator determine the full hash node representation. Refer to the defined structures H_NODE_INT and H_NODE_STRING for examples of the general purpose hash nodes used by the hashing functions and hash node comparators defined in this library.

HASHING FUNCTIONS

One function, called the hashing function, controls the distribution of nodes in the table. This library provides a number of standard hashing functions, but applications can specify their own. Desirable properties of a hashing function are that they execute quickly, and evenly distribute the nodes throughout the table. The worst hashing function imaginable would be: $h(k) = 0$. This function would put all nodes in a list associated with the zero element in the hash table. Most hashing functions find their origin in random number generators.

Hashing functions must return an index between zero and (elements - 1). They take the following form:

```
int hashFuncXXX
(
    int          elements,      /* number of elements in hash table
*/
    HASH_NODE *  pHashNode,    /* hash node to pass through hash function */
    int          keyArg        /* optional argument to hash function
*/
)
```

HASH NODE COMPARATOR FUNCTIONS

The second function required is a key comparator. Different hash tables may choose to compare hash nodes in different ways. For example, the hash node could contain a key which is a pointer to a string, or simply an integer. The comparator compares the hash node on the basis of some criteria, and returns a boolean as to the nodes equivalence. Additionally, the key comparator can use the keyCmpArg for additional information to the comparator. The keyCmpArg is passed from all the **hashLib** functions which use the comparator. The keyCmpArg is usually not needed except for advanced hash table querying.

symLib is a good example of the utilization of the keyCmpArg parameter. **symLib** hashes the name of the symbol. It finds the id based on the name using **hashTblFind()**, but for the purposes of putting and removing symbols from the symbol's hash table, an additional comparison restriction applies. Symbols have types, and while symbols of equivalent names can exist, no symbols of equivalent name and type can exist. So **symLib** utilizes the keyCmpArg as a flag to denote which operation is being performed on the hash table: symbol name matching, or complete symbol name and type matching.

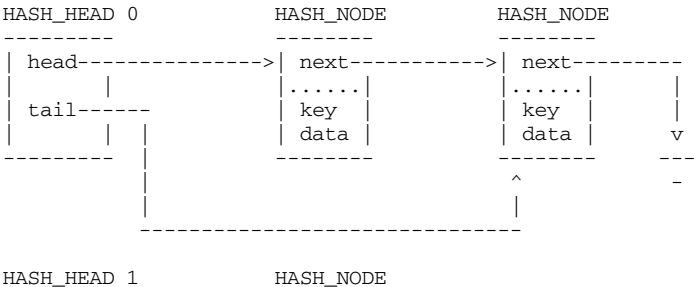
Key comparator functions must return a boolean. They take the following form:

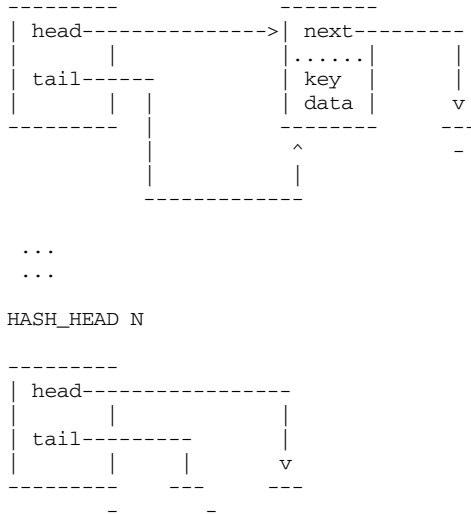
```
BOOL hashKeyCmpXXX
(
    HASH_NODE * pMatchNode, /* hash node to match */
    HASH_NODE * pHashNode, /* hash node in table being compared to */
    int         keyCmpArg   /* parameter passed to hashTblFind (2) */
)
```

HASHING COLLISIONS

Hashing collisions occur when the hashing function returns the same index when given two unique keys. This is unavoidable in cases where there are more nodes in the hash table than there are elements in the hash table. In a chained hash table, collisions are resolved by treating each element of the table as the head of a linked list. Nodes are simply added to an appropriate list regardless of other nodes already in the list. The list is not sorted, but new nodes are added at the head of the list because newer entries are usually searched for before older entries. When nodes are removed or searched for, the list is traversed from the head until a match is found.

STRUCTURE





HASH_HEAD N

CAVEATS	Hash tables must have a number of elements equal to a power of two.
INCLUDE FILE	<code>hashLib.h</code>

hookLib

NAME	hookLib – generic hook library for VxWorks
------	--

ROUTINES	hookAddToTail() – add a hook routine to the end of a hook table
	hookAddToHead() – add a hook routine at the start of a hook table
	hookDelete() – delete a hook from a hook table
	hookFind() – Search a hook table for a given hook

DESCRIPTION	This library provides generic functions to add and delete hooks. Hooks are function pointers, that when set to a non-NULL value are called by VxWorks at specific points in time. The hook primitives provided by this module are used by many VxWorks facilities such as taskLib , rtpLib , syscallLib etc.
--------------------	---

A hook table is an array of function pointers. The size of the array is decided by the various facilities using this library. The head of a hook table is the first element in the table (i.e. offset 0), while the tail is the last element (i.e. highest offset). Hooks can be added either to the head or the tail of a given hook table. When added to the tail, a new routine is added after the last non-NULL entry in the table. When added to the head of a table, new routines are

added at the head of the table (index 0) after existing routines have been shifted down to make room.

Hook execution always proceeds starting with the head (index 0) till a NULL entry is reached. Thus adding routines to the head of a table achieves a LIFO-like effect where the most recently added routine is executed first. In contrast, routines added to the tail of a table are executed in the order in which they were added. For example, task creation hooks are examples of hooks added to the tail, while task deletion hooks are an example of hooks added to the head of their respective table. Hook execution macros **HOOK_INVOKE_VOID_RETURN** and **HOOK_INVOKE_CHECK_RETURN** (defined in **hookLib.h**) are handy in calling hook functions. Alternatively, users may write their own invocations.

NOTE It is possible to have dependencies among hook routines. For example, a delete hook may use facilities that are cleaned up and deleted by another delete hook. In such cases, the order in which the hooks run is important. VxWorks runs the create and switch hooks in the order in which they were added, and runs the delete hooks in reverse of the order in which they were added. Thus, if the hooks are added in "hierarchical" order, such that they rely only on facilities whose hook routines have already been added, then the required facilities will be initialized before any other facilities need them, and will be deleted after all facilities are finished with them.

VxWorks facilities guarantee this by having each facility's initialization routine first call any prerequisite facility's initialization routine before adding its own hooks. Thus, the hooks are always added in the correct order. Each initialization routine protects itself from multiple invocations, allowing only the first invocation to have any effect.

INCLUDE FILES **hookLib.h**

SEE ALSO **dbgLib**, **taskLib**, **taskVarLib**, **rtpLib**, the VxWorks programmer, guides.

hookShow

NAME **hookShow** – hook show routines

ROUTINES **hookShow()** – show the hooks in the given hook table

DESCRIPTION This library provides routines which summarize the installed kernel hook routines in a given hook table. These routines are generic, and can be used to display any kind of hooks. To include this library, select the **INCLUDE_HOOK_SHOW** component.

INCLUDE FILES **hookLib.h**

SEE ALSO **hookLib**, "VxWorks Kernel Programmer's Guide: Basic OS"

hrFsLib

NAME	hrFsLib – highly reliable file system library
ROUTINES	hrfsDevCreate() – create an HRFS device
DESCRIPTION	This library contains routines for creating and using the Highly Reliable File System (HRFS).
INCLUDE FILES	hrFsLib.h

hrFsTimeLib

NAME	hrFsTimeLib – time routines for HRFS
ROUTINES	hrfsTimeGet() – return # of milliseconds since midnight Jan 1, 1970 hrfsAscTime() – convert "broken-down" HRFS time to string hrfsTimeSplit() – split time in msec into HRFS_TM format hrfsTimeCondense() – condense time in HRFS_TM to time in msec
DESCRIPTION	This library contains routines for handling the HRFS timestamps.
INCLUDE FILES	none

hrfsChkDskLib

NAME	hrfsChkDskLib – HRFS Check disk library - Readonly version
ROUTINES	hrfsChkDsk() – check the HRFS file system hrfsUpgrade() – upgrade the HRFS file system to the latest version
DESCRIPTION	This library contains routines for the Highly Reliable File System (HRFS) consistency disk checker or check disk. This is a read-only utility in that it does not attempt to correct any errors it detects but simply reports them.

INCLUDE FILES none

hrfsFormatLib

NAME	hrfsFormatLib – HRFS format library
ROUTINES	hrfsFormatLibInit() – prepare to use the HRFS formatter hrfsFormatFd() – format the HRFS file system via a file descriptor hrfsAdvFormatFd() – format the HRFS file system using advanced options via a file descriptor hrfsFormat() – format the HRFS file system via a path hrfsAdvFormat() – format the HRFS file system using advanced options
DESCRIPTION	This library contains routines for formatting the Highly Reliable File System (HRFS).
INCLUDE FILES	none

inflateLib

NAME	inflateLib – inflate code using public domain zlib functions
ROUTINES	inflate() – inflate compressed code
DESCRIPTION	<p>This library is used to inflate a compressed data stream, primarily for boot ROM decompression. Compressed boot ROMs contain a compressed executable in the data segment between the symbols binArrayStart and binArrayEnd (the compressed data is generated by deflate() and binToAsm). The boot ROM startup code (in target/src/config/all/bootInit.c) calls inflate() to decompress the executable and then jump to it.</p> <p>This library is based on the public domain zlib code, which has been modified by Wind River Systems. For more information, see the zlib home page at http://www.gzip.org/zlib/.</p>

OVERVIEW OF THE COMPRESSION/DECOMPRESSION

1. Compression algorithm (deflate)
- The deflation algorithm used by zlib (also zip and gzip) is a variation of LZ77 (Lempel-Ziv 1977, see reference below). It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair

(distance, length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. (In this description, **string** must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. The blocks can have any size (except that the compressed data for one block must fit in available memory). A block is terminated when **deflate()** determines that it would be useful to start another block with fresh trees. (This is somewhat similar to the behavior of LZW-based `_compress_`.)

Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option (level parameter of `deflateInit`). So **deflate()** does not always find the longest possible match but generally finds a match which is long enough.

deflate() also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, **deflate()** searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards. Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a runtime parameter. If the current match is long enough, **deflate()** reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, **deflate()** attempts a complete second search even if the first match is already long enough.

The lazy match evaluation is not performed for the fastest compression modes (level parameter 1 to 3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

2. Decompression algorithm (zinflate)

The real question is, given a Huffman tree, how to decode fast. The most important realization is that shorter codes are much more common than longer codes, so pay attention to decoding the short codes fast, and let the long codes take longer to decode.

zinflate() sets up a first level table that covers some number of bits of input less than the length of longest code. It gets that many bits from the stream, and looks it up in the table. The table will tell if the next code is that many bits or less and how many, and if it is, it will tell the value, else it will point to the next level table for which **zinflate()** grabs more bits and tries to decode a longer code.

How many bits to make the first lookup is a tradeoff between the time it takes to decode and the time it takes to build the table. If building the table took no time (and if you had infinite memory), then there would only be a first level table to cover all the way to the longest code. However, building the table ends up taking a lot longer for more bits since short codes are replicated many times in such a table. What **zinflate()** does is simply to make the number of bits in the first table a variable, and set it for the maximum speed.

zinflate() sends new trees relatively often, so it is possibly set for a smaller first level table than an application that has only one tree for all the data. For **zinflate**, which has 286 possible codes for the literal/length tree, the size of the first table is nine bits. Also the distance trees have 30 possible values, and the size of the first table is six bits. Note that for each of those cases, the table ended up one bit longer than the **average** code length, i.e. the code length of an approximately flat code which would be a little more than eight bits for 286 symbols and a little less than five bits for 30 symbols. It would be interesting to see if optimizing the first level table for other applications gave values within a bit or two of the flat code size.

Jean-loup Gailly Mark Adler gzip@prep.ai.mit.edu madler@alumni.caltech.edu

References:

[LZ77] Ziv J., Lempel A., 'A Universal Algorithm for Sequential Data Compression,' IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

DEFLATE Compressed Data Format Specification available in
<ftp://ds.internic.net/rfc/rfc1951.txt>

MORE INTERNAL DETAILS

Huffman code decoding is performed using a multi-level table lookup. The fastest way to decode is to simply build a lookup table whose size is determined by the longest code. However, the time it takes to build this table can also be a factor if the data being decoded is not very long. The most common codes are necessarily the shortest codes, so those codes dominate the decoding time, and hence the speed. The idea is you can have a shorter table that decodes the shorter, more probable codes, and then point to subsidiary tables for the longer codes. The time it costs to decode the longer codes is then traded against the time it takes to make longer tables.

This results of this trade are in the variables **lbits** and **dbits** below. **lbits** is the number of bits the first level table for literal/ length codes can decode in one step, and **dbits** is the same thing for the distance codes. Subsequent tables are also less than or equal to those sizes. These values may be adjusted either when all of the codes are shorter than that, in which case the longest code length in bits is used, or when the shortest code is *longer* than the requested table size, in which case the length of the shortest code in bits is used.

There are two different values for the two tables, since they code a different number of possibilities each. The literal/length table codes 286 possible values, or in a flat code, a little over eight bits. The distance table codes 30 possible values, or a little less than five bits, flat. The optimum values for speed end up being about one bit more than those, so lbits is 8+1 and dbits is 5+1. The optimum values may differ though from machine to machine, and possibly even between compilers. Your mileage may vary.

Notes beyond the 1.93a **appnote.txt**:

1. Distance pointers never point before the beginning of the output stream.
2. Distance pointers can point back across blocks, up to 32k away.
3. There is an implied maximum of 7 bits for the bit length table and 15 bits for the actual data.
4. If only one code exists, then it is encoded using one bit. (Zero would be more efficient, but perhaps a little confusing.) If two codes exist, they are coded using one bit each (0 and 1).
5. There is no way of sending zero distance codes--a dummy must be sent if there are none. (History: a pre 2.0 version of PKZIP would store blocks with no distance codes, but this was discovered to be too harsh a criterion.) Valid only for 1.93a. 2.04c does allow zero distance codes, which is sent as one code of zero bits in length.
6. There are up to 286 literal/length codes. Code 256 represents the end-of-block. Note however that the static length tree defines 288 codes just to fill out the Huffman codes. Codes 286 and 287 cannot be used though, since there is no length base or extra bits defined for them. Similarly, there are up to 30 distance codes. However, static trees define 32 codes (all 5 bits) to fill out the Huffman codes, but the last two had better not show up in the data.
7. Unzip can check dynamic Huffman blocks for complete code sets. The exception is that a single code would not be complete (see #4).
8. The five bits following the block type is really the number of literal codes sent minus 257.
9. Length codes 8,16,16 are interpreted as 13 length codes of 8 bits (1+6+6). Therefore, to output three times the length, you output three codes (1+1+1), whereas to output four times the same length, you only need two codes (1+3). Hmm.
10. In the tree reconstruction algorithm, Code = Code + Increment only if BitLength(i) is not zero. (Pretty obvious.)
11. Correction: 4 Bits: # of Bit Length codes - 4 (4 - 19)
12. Note: length code 284 can represent 227-258, but length code 285 really is 258. The last length deserves its own, short code since it gets used a lot in very redundant files. The length 258 is special since 258 - 3 (the min match length) is 255.

13. The literal/length and distance code bit lengths are read as a single stream of lengths. It is possible (and advantageous) for a repeat code (16, 17, or 18) to go across the boundary between the two sets of lengths.

INCLUDE FILES none

intArchLib

NAME `intArchLib` – architecture-dependent interrupt library

ROUTINES

- `intLevelSet()` – set the interrupt level (MC680X0, x86, ARM, SimSolaris, SimNT and SH)
- `intLock()` – lock out interrupts
- `intUnlock()` – cancel interrupt locks
- `intCpuLock()` – lock out interrupts on local CPU
- `intCpuUnlock()` – cancel local CPU interrupt lock
- `intEnable()` – enable corresponding interrupt bits (MIPS, PowerPC, ARM)
- `intDisable()` – disable corresponding interrupt bits (MIPS, PowerPC, ARM)
- `intCRGet()` – read the contents of the cause register (MIPS)
- `intCRSet()` – write the contents of the cause register (MIPS)
- `intSRGet()` – read the contents of the status register (MIPS)
- `intSRSet()` – update the contents of the status register (MIPS)
- `intConnect()` – connect a C routine to a hardware interrupt
- `intHandlerCreate()` – construct an interrupt handler for a C routine (MC680x0, x86, MIPS, SimSolaris)
- `intLockLevelSet()` – set the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT)
- `intLockLevelGet()` – get the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT)
- `intVecBaseSet()` – set the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT)
- `intVecBaseGet()` – get the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT)
- `intVecSet()` – set a CPU vector (trap) (MC680x0, x86, MIPS, SH, SimSolaris, SimNT)
- `intVecGet()` – get an interrupt vector (MC680x0, x86, MIPS, SH, SimSolaris, SimNT)
- `intVecTableWriteProtect()` – write-protect exception vector table (MC680x0, x86, ARM, SimSolaris, SimNT)
- `intUninitVecSet()` – set the uninitialized vector handler (ARM)
- `intHandlerCreateI86()` – construct an interrupt handler for a C routine (x86)
- `intVecSet2()` – set a CPU vector, gate type(int/trap), and selector (x86)
- `intVecGet2()` – get a CPU vector, gate type(int/trap), and gate selector (x86)
- `intStackEnable()` – enable or disable the interrupt stack usage (x86)

DESCRIPTION This library provides architecture-dependent routines to manipulate and connect to hardware interrupts. Any C language routine can be connected to any interrupt by calling **intConnect()**. Vectors can be accessed directly by **intVecSet()** and **intVecGet()**. The vector (trap) base register (if present) can be accessed by the routines **intVecBaseSet()** and **intVecBaseGet()**.

Tasks can lock and unlock interrupts by calling **intLock()**, **intCpuLock()**, **intUnlock()** and **intCpuUnlock()**. The lock-out level can be set and reported by **intLockLevelSet()** and **intLockLevelGet()** (MC680x0, x86, ARM and SH only). The routine **intLevelSet()** changes the current interrupt level of the processor (MC680x0, ARM, SimSolaris and SH).

LOCAL CPU INTERRUPT LOCKING

The VxWorks SMP kernel does not allow locking of interrupts on all CPUs that make up the system. Therefore the **intLock()** and **intUnlock()** APIs are not available in VxWorks SMP. However interrupt locking is permitted on the local CPU, which is the CPU the task or ISR is running on when it calls **intCpuLock()** to lock interrupts. The **intCpuUnlock()** routine is used to re-enable interrupts on the local CPU. The **intCpuLock()/intCpuUnlock()** pair is available on the uniprocessor version of VxWorks but since the local CPU is the only CPU in that situation, the behaviour of these routines is identical to the behaviour of the **intLock()/intUnlock()** routines.

INVOKING VxWorks SYSTEM ROUTINES WITH INTERRUPTS LOCKED

Invoking a VxWorks system routine after having locked interrupts using **intLock()** may result in interrupts being re-enabled for an unspecified period of time. See the reference entry for **intLock()** for more details.

Invoking a VxWorks system routine after having locked interrupts using **intCpuLock()** on VxWorks SMP is not permitted and will cause the call to abort and an error to be reported. Not all VxWorks APIs enforce this restriction. Only those that are *intCpuLock restricted*. The reference entries in the VxWorks Kernel API Reference manual specifies when this restriction applies. Since the **intCpuLock()** behaviour in the uniprocessor version of VxWorks is identical to the **intLock()** API behaviour, the concept of *intCpuLock restricted* APIs only applies to VxWorks SMP.

INTERRUPT VECTORS AND NUMBERS

Most of the routines in this library take an interrupt vector as a parameter, which is generally the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers:

IVEC_TO_INUM(intVector) 10
converts a vector to a number.

INUM_TO_IVEC(intNumber)
converts a number to a vector.

TRAPNUM_TO_IVEC(trapNumber)
converts a trap number to a vector.

EXAMPLE To switch between one of several routines for a particular interrupt, the following code fragment is one alternative:

```
vector = INUM_TO_IVEC(some_int_vec_num);
oldfunc = intVecGet (vector);
newfunc = intHandlerCreate (routine, parameter);
intVecSet (vector, newfunc);
...
intVecSet (vector, oldfunc);    /* use original routine */
...
intVecSet (vector, newfunc);    /* reconnect new routine */
```

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **iv.h, intLib.h**

SEE ALSO **intLib**

intLib

NAME **intLib** – architecture-independent interrupt subroutine library

ROUTINES **intContext()** – determine if executing in interrupt context
intCount() – get the current interrupt nesting depth
intDisconnect() – disconnect a C routine from a hardware interrupt

DESCRIPTION This library provides generic routines for interrupts. Any C language routine can be connected (disconnect) to (from) any interrupt (trap) by calling **intConnect()** (**intDisconnect()**), which resides in **intArchLib**. The **intCount()** and **intContext()** routines are used to determine whether the CPU is running in an interrupt context or in a normal task context. For information about architecture-dependent interrupt handling, see the reference entry for **intArchLib**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the

ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

CONFIGURATION	The interrupt subroutine library is always included in the VxWorks kernel.
INCLUDE FILES	intLib.h
SEE ALSO	intArchLib , The VxWorks Programmer Guides.

ioLib

NAME	ioLib – I/O interface library
ROUTINES	creat() – create a file open() – open a file close() – close a file read() – read bytes from a file or device write() – write bytes to a file lseek() – set a file read/write pointer ioctl() – perform an I/O control function ioGlobalStdSet() – set file descriptor for global input/output/error ioGlobalStdGet() – get the file descriptor for global input/output/error ioTaskStdSet() – set the file descriptor for task standard input/output/error ioTaskStdGet() – get the file descriptor for task standard input/output/error isatty() – return whether the underlying driver is a <i>tty</i> device fcntl() – perform control functions over open files
DESCRIPTION	<p>This library contains the interface to the basic I/O system. It includes:</p> <ul style="list-style-type: none">- Interfaces to the seven basic driver-provided functions: creat(), remove(), open(), close(), read(), write(), and ioctl().- Interfaces to several file system functions, including rename() and lseek().- Routines to set and get the current working directory.- Routines to assign task and global standard file descriptors.

FILE DESCRIPTORS

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

Three file descriptors are reserved and have special meanings:

```
0 (`STD_IN') - standard input
1 (`STD_OUT') - standard output
2 (`STD_ERR') - standard error output
```

VxWorks allows two levels of redirection. First, there is a global assignment of the three standard file descriptors. By default, new tasks use this global assignment. The global assignment of the three standard file descriptors is controlled by the routines **ioGlobalStdSet()** and **ioGlobalStdGet()**.

Second, individual tasks may override the global assignment of these file descriptors with their own assignments that apply only to that task. The assignment of task-specific standard file descriptors is controlled by the routines **ioTaskStdSet()** and **ioTaskStdGet()**.

CONFIGURATION	To include the I/O interface library, configure VxWorks with the INCLUDE_IO_SYSTEM component.
INCLUDE FILES	ioLib.h , stdio.h
SEE ALSO	iosLib , ansiStdio , the VxWorks programmer guides.

iosLib

NAME	iosLib – I/O system library
ROUTINES	iosInit() – initialize the kernel I/O system iosDrvInstall() – install a kernel I/O driver iosDevAdd() – add a device to the kernel I/O system iosDevDelete() – delete a device from the kernel I/O system iosDevDelDrv() – invoke device delete driver if reference counter reaches 0. iosDrvRemove() – remove a kernel I/O driver iosDevFind() – find an I/O device in the kernel device list iosFdMaxFiles() – return maximum files for current RTP iosFdEntryGet() – get an unused FD_ENTRY from the pool iosFdEntryReturn() – return an FD_ENTRY to the pool
DESCRIPTION	<p>This library is the driver-level interface to the I/O system. Its primary purpose is to route user I/O requests to the proper drivers, using the proper parameters. To do this, iosLib keeps tables describing the available drivers (e.g., names, open files).</p> <p>The I/O system should be initialized by calling iosInit(), before calling any other routines in iosLib. Each driver then installs itself by calling iosDrvInstall(). The devices serviced by each driver are added to the I/O system with iosDevAdd().</p>

The I/O system is described more fully in the VxWorks programmer guides.

CONFIGURATION	To use the driver-level I/O interface, configure VxWorks with the INCLUDE_IO_SYSTEM component.
INCLUDE FILES	iosLib.h
SEE ALSO	intLib , ioLib , the VxWorks programmer guides.

iosShow

NAME	iosShow – I/O system show routines
ROUTINES	iosShowInit() – initialize the I/O system show facility iosDrvShow() – display a list of system drivers iosDevShow() – display the list of devices in the system iosFdShow() – display a list of file descriptor names in the system iosRtpFdShow() – show the per-RTP <i>fd</i> table
DESCRIPTION	This library contains I/O system information display routines. The routine iosShowInit() links the I/O system information show facility into the VxWorks system. It is called automatically when the I/O show routines are included.
CONFIGURATION	To use the I/O system show routines, configure VxWorks with the INCLUDE_IO_SYSTEM and INCLUDE_SHOW_ROUTINES components.
INCLUDE FILES	ioLib.h , stdio.h
SEE ALSO	intLib , ioLib , windsh , the VxWorks programmer guides, and the IDE and host tools guides.

isrLib

NAME	isrLib – isr objects library
ROUTINES	isrCreate() – create an ISR object isrDelete() – delete an ISR object isrInvoke() – invoke the handler routine of an ISR object isrIdSelf() – get the ISR ID of the currently running ISR

isrInfoGet() – get information about an ISR object

DESCRIPTION

This library contains routines to manage ISR objects. Specifically it provides the ability to create, delete and obtain information about ISR objects. It can also be used to invoke the handler associated with an ISR object and to determine which ISR is currently being processed. There is no configuration component associated with ISR objects since the functionality is always present in the VxWorks kernel.

This library is not a replacement for **intLib**. It is complementary to it and is in fact used by **intLib** to create ISR objects when a routine is connected to an interrupt vector via **intConnect()** as explained in more details below.

ISR objects are WIND object hence they can also be managed using the **objLib** API. For example, the name of an ISR object can be obtained using **objNameGet()**.

The vast majority of users need not be concerned with the **isrCreate()**, **isrDelete()** and **isrInvoke()** routines. These are meant to be used by interrupt controller drivers and BSPs that use chaining or multiplexing of interrupts so that a true representation of the interrupt architecture can be maintained by this library.

Creation of ISR Objects

Can be done in one of two ways:

1) Implicit creation by calling **intConnect()**

This creation method allows an interrupt service routine provider to not be concerned with the creation of an ISR object since one is automatically created when the routine is connected to a vector using **intConnect()**. Recall that rules apply regarding how early in the booting sequence **intConnect()** can be called. Refer to the BSP Developer's Guide for more details on this subject.

1) Explicitly calling **isrCreate()**

This creation method is meant to be used by code that connects interrupt service routines to vectors using a routine other than **intConnect()**. For example, an auxiliary clock driver that connects a handler to the programmable interval timer exception on PPC using **excIntConnect()** would need to explicitly create an ISR object to represent the handler otherwise this library would be unaware of its existence. Failure to create an ISR object under these circumstances does not affect the ability of the system to handle interrupts. It simply causes a discrepancy between the actual interrupt architecture of the system and the representation **isrLib** has of this architecture.

Destruction of ISR Objects

The destruction model is very similar to the creation model in that an **isrDelete()** call is performed automatically when **intDisconnect()** is called. Code which disconnects an interrupt service routine from a vector by other means must ensure **isrDelete()** is called to delete the associated ISR object.

Invocation of an Interrupt Handler

The creation process of an ISR object requires that a handler be specified. This information is stored in the ISR object itself and the handler is automatically invoked when the associated interrupt occurs if the creation of the ISR object was done implicitly. That is, done through **intConnect()** as described above. However, in the case of an ISR object that is explicitly created, the creator must arrange for the **isrInvoke()** routine to be called when the associated interrupt occurs. This can be done by installing **isrInvoke()** as the interrupt handler or by having the interrupt handler call **isrInvoke()** directly. Routine **isrInvoke()** then ensures the handler associated with the ISR object is invoked. See the coding example below for more details on this subject.

Obtaining Information about ISR Objects

Routine **isrInfoGet()** allows one to obtain information about a specific ISR object. Routine **isrShow()**, which is provided by and documented in library **isrShow**, can also be used for information gathering purposes.

Determining the Currently Running ISR

Routine **isrIdSelf()** allows the calling ISR to determine its ID. This is similar in principle to routine **taskIdSelf()**.

CODING EXAMPLE

This example illustrates how a PPC-based BSP can make use of **isrLib** to create an ISR object when connecting the auxiliary clock handler to the **_EXC_OFF_FIT** exception. The advantage of doing so is that the interrupt becomes visible to the system when **isrShow()** is used for example.

The reason **sysHwInit2()** has to explicitly create an ISR object to represent the auxiliary clock interrupt is because the handler is connected by a means other than **intConnect()**. In this case routine **excIntConnect()** is used.

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

/* Forward declarations */
LOCAL void sysBaseAuxClkInt (void);

/* Externs */
extern void sysAuxClkInt (void);

/* Local variables */
LOCAL ISR_ID auxClkIsrId;

void sysHwInit2 (void)
{
    static BOOL configured = FALSE;

    if (!configured)
    {
```

```
/*
 * Create an ISR object specifying <sysAuxClkInt> as the <handlerRtn>
 * and 0 as the <parameter>. This is because sysAuxClkInt()
 * does not expect an argument.
 */
auxClkIsrId = isrCreate ("sysAuxClk", 0, (FUNCPTR) &sysAuxClkInt,
                        0, 0);

if (auxClkIsrId != NULL)
{
    /*
     * ISR object created successfully. Install wrapper routine as
     * the FIT exception handler.
     */
    excIntConnect ((VOIDFUNCPTR *) _EXC_OFF_FIT, &sysBaseAuxClkInt);
}
else
{
    /*
     * ISR object creation failed. Install sysAuxClkInt() directly as
     * the FIT exception handler. This is how it was done before
     * ISR objects came to existence.
     */
    excIntConnect ((VOIDFUNCPTR *) _EXC_OFF_FIT, &sysAuxClkInt);
}

/* Do other types of hardware initialization */

configured = TRUE;
}
}

LOCAL void sysBaseAuxClkInt (void)
{
    isrInvoke (auxClkIsrId);
}
```

INCLUDE FILES **isrLib.h**

SEE ALSO **intLib**

isrShow

NAME **isrShow** – isr objects show library

ROUTINES **isrShow()** – show information about an ISR object

DESCRIPTION This library provides the routine **isrShow()** to show the contents of ISR objects.

CONFIGURATION	The routines in this library are included if the INCLUDE_ISR_SHOW component is configured into VxWorks.
INCLUDE FILES	none
SEE ALSO	isrLib

kern_sysctl

NAME	kern_sysctl – sysctl kernel routines
ROUTINES	sysctl_remove_oid() – remove dynamically created sysctl trees sysctl_add_oid() – add a parameter into the sysctl tree during run-time sysctl() – get or set the the values of objects in the sysctl tree sysctlbyname() – get or set the values of objects in the sysctl tree by name sysctlnametomib() – return the numeric representation of sysctl object
DESCRIPTION	<p>This module contains the definitions of various sysctl related functions. Although, there are a number of functions in this module, the ones that are are of significant importance are sysctl() and sysctlbyname().</p> <p>sysctl() and sysctlbyname() basically accomplish the same task. sysctlbyname() is a more user friendly routine. sysctl() expects the caller to have mapped the name of the variable to the corresponding sysctl MIB OID. sysctlbyname() expects the caller to provide the name of the variable (including the path in dot format) and maps it internally to the OID.</p>
INCLUDE FILES	sys/sysctl.h

kernelLib

NAME	kernelLib – VxWorks kernel library
ROUTINES	kernelInit() – initialize the kernel kernelVersion() – return the WIND kernel revision string kernelTimeSlice() – enable round-robin selection kernelRoundRobinInstall() – install VxWorks Round Robin implementation kernelCpuEnable() – enable a CPU kernelIsCpuIdle() – determine whether the specified CPU is idle kernelIsSystemIdle() – determine whether all enabled processors are idle

DESCRIPTION The VxWorks kernel provides tasking control services to an application. The libraries **kernelLib**, **taskLib**, **semLib**, **tickLib**, and **wdLib** comprise the kernel functionality. This library is the interface to the VxWorks kernel initialization, revision information, and scheduling control.

KERNEL INITIALIZATION

The kernel must be initialized before any other kernel operation is performed. Normally kernel initialization is taken care of by the system configuration code in **usrInit()** in **usrConfig.c** or **prjConfig.c**.

Kernel initialization consists of the following:

- (1) Defining the starting address and size of the system memory partition. The **malloc()** routine uses this partition to satisfy memory allocation requests of other facilities in VxWorks.
- (2) Allocating the specified memory size for an interrupt stack. Interrupt service routines will use this stack unless the underlying architecture does not support a separate interrupt stack, in which case the service routine will use the stack of the interrupted task.
- (3) Specifying the interrupt lock-out level. VxWorks will not exceed the specified level during any operation. The lock-out level is normally defined to mask the highest priority possible. However, in situations where extremely low interrupt latency is required, the lock-out level may be set to ensure timely response to the interrupt in question. Interrupt service routines handling interrupts of priority greater than the interrupt lock-out level may not call any VxWorks routine.

Once the kernel initialization is complete, a root task is spawned with the specified entry point and stack size. The root entry point is normally **usrRoot()** of the **usrConfig.c** or **prjConfig.c** module. The remaining VxWorks initialization takes place in **usrRoot()**.

ROUND-ROBIN SCHEDULING

Round-robin scheduling allows the processor to be shared fairly by all tasks of the same priority. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single non-blocking task can usurp the processor until preempted by a task of higher priority, thus never giving the other equal-priority tasks a chance to run.

Round-robin scheduling is disabled by default. It can be enabled or disabled with the routine **kernelTimeSlice()**, which takes a parameter for the "time slice" (or interval) that each task will be allowed to run before relinquishing the processor to another equal-priority task. If the parameter is zero, round-robin scheduling is turned off. If round-robin scheduling is enabled and preemption is enabled for the executing task, the system tick handler will increment 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. New tasks 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 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 is eligible for execution. In the case of preemption, the task will resume execution once the higher priority task completes, assuming no other task of a higher priority is ready to run. For the case when the task blocks, it is placed at the tail of the list of tasks at its priority. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

Time-slice counts are accrued against the task that is executing when a system tick occurs regardless of whether the task has executed for the entire tick interval. Due to preemption by higher priority tasks or ISRs stealing CPU time from the task, scenarios exist where a task can execute for less or more total CPU time than it's allotted time slice.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **kernelLib.h**

SEE ALSO **taskLib**, **intLib**, *"VxWorks Kernel Programmer's Guide: Basic OS"*

ledLib

NAME **ledLib** – line-editing library

ROUTINES **ledLibInit()** – initialize the line editing facilities
ledOpen() – create a new line-editor ID
ledClose() – discard the line-editor ID
ledRead() – read a line with line-editing
ledControl() – change the line-editor ID parameters

DESCRIPTION This library provides a line-editing layer on top of a **tty** device. The shell uses this interface for its history-editing features.

The editing mode of the shell can be configured using the project tool:

- vi-like editing mode (`INCLUDE_SHELL_VI_MODE`)
- emacs-like editing mode (`INCLUDE_SHELL_EMACS_MODE`)

VI-LIKE EDITING MODE

The shell history mechanism is similar to the UNIX Korn shell history facility, with a built-in line-editor similar to UNIX `vi` that allows previously typed commands to be edited. The command `h()` displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, type `ESC` to enter edit mode, and use the commands listed below. The `ESC` key switches the shell to edit mode. The `RETURN` key always gives the line to the shell from either editing or input mode.

The following list is a summary of the commands available in edit mode.

Movement and search commands

<code>nG</code>	Go to command number <i>n</i> .
<code>/s</code>	Search for string <i>s</i> backward in history.
<code>?s</code>	Search for string <i>s</i> forward in history.
<code>n</code>	Repeat last search.
<code>N</code>	Repeat last search in opposite direction.
<code>nk</code>	Get <i>n</i> th previous shell command in history.
<code>n-</code>	Same as "k".
<code>nj</code>	Get <i>n</i> th next shell command in history.
<code>n+</code>	Same as "j".
<code>nh</code>	Move left <i>n</i> characters.
<code>CTRL-H</code>	Same as "h".
<code>nl</code>	Move right <i>n</i> characters.
<code>SPACE</code>	Same as "l".
<code>nw</code>	Move <i>n</i> words forward.
<code>nW</code>	Move <i>n</i> blank-separated words forward.
<code>ne</code>	Move to end of the <i>n</i> th next word.
<code>nE</code>	Move to end of the <i>n</i> th next blank-separated word.
<code>nb</code>	Move back <i>n</i> words.
<code>nB</code>	Move back <i>n</i> blank-separated words.
<code>fc</code>	Find character <i>c</i> , searching forward.
<code>Fc</code>	Find character <i>c</i> , searching backward.
<code>^</code>	Move cursor to first non-blank character in line.
<code>\$</code>	Go to end of line.
<code>0</code>	Go to beginning of line.
	Insert commands (input is expected until an <code>ESC</code> is typed).
<code>a</code>	Append.
<code>A</code>	Append at end of line.
<code>c SPACE</code>	Change character.
<code>cl</code>	Change character.
<code>cw</code>	Change word.

Movement and search commands

cc	Change entire line.
c\$	Change everything from cursor to end of line.
C	Same as "c\$".
S	Same as "cc".
s	Same as "cl".
i	Insert.
I	Insert at beginning of line.
R	Type over characters.

Editing commands

<i>nrc</i>	Replace the following <i>n</i> characters with <i>c</i> .
<i>nx</i>	Delete <i>n</i> characters starting at cursor.
<i>nX</i>	Delete <i>n</i> characters to the left of the cursor.
d SPACE	Delete character.
dl	Delete character.
dw	Delete word.
dd	Delete entire line.
d\$	Delete everything from cursor to end of line.
D	Same as "d\$".
p	Put last deletion after the cursor.
P	Put last deletion before the cursor.
u	Undo last command.
~	Toggle case, lower to upper or vice versa.

Special commands

CTRL+U	Delete line and leave edit mode.
CTRL+L	Redraw line.
CTRL+D	Complete symbol name.
RETURN	Give line to the shell and leave edit mode.

The default value for *n* is 1.

DEFICIENCIES Since the shell toggles between raw mode and line mode, type-ahead can be lost. The ESC, redraw, and non-printable characters are built-in.

Some commands do not take counts as users might expect. For example, "*ni*" will not insert whatever was entered *n* times.

EMACS-LIKE EDITING MODE

The shell history mechanism is similar to the UNIX Tcsh shell history facility, with a built-in line-editor similar to **emacs** that allows previously typed commands to be edited. The command **h()** displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, the arrow keys can be used on most of the terminals. Up arrow and down arrow move up and down through the history list, like CTRL+P and CTRL+N. Left arrow and right arrow move the cursor left and right one character, like CTRL+B and CTRL+F.

The following list is a summary of the commands available with the emacs-like editing mode.

Cursor motion commands	
CTRL+B	Move cursor back (left) one character.
CTRL+F	Move cursor forward (right) one character.
ESC+b	Move cursor back one word.
ESC+f	Move cursor forward one word.
CTRL+A	Move cursor to beginning of line.
CTRL+E	Move cursor to end of line.
Modification commands	
DEL or	Delete character to left of cursor.
CTRL+H	
CTRL+D	Delete character under cursor.
ESC+d	Delete word.
ESC+DEL	Delete word backward.
CTRL+K	Delete from cursor to end of line.
CTRL+U	Delete entire line.
CTRL+P	Get previous command in the history.
CTRL+N	Get next command in the history.
!n	Recall command <i>n</i> from the history.
!substr	Recall first command from the history matching <i>substr</i> .
Special commands	
CTRL+L	Redraw line.
CTRL+D	Complete symbol name if cursor at the end of line.
RETURN	Give line to the shell.

INCLUDE FILES **ledLib.h**

SEE ALSO *VxWorks Kernel Programmer's Guide: **Kernel Shell**, Wind River Workbench Command-Line User's Guide 2.2: **Host Shell***

loadLib

NAME	loadLib – generic object module loader
ROUTINES	loadModule() – load an object module into memory loadModuleAt() – load an object module into memory
DESCRIPTION	This library provides a generic object module loading facility. It handles loading ELF format files into memory, relocating them, resolving their external references, and adding their external definitions to the system symbol table for use by other modules and from the shell.

Modules may be loaded from any I/O stream which allows repositioning of the pointer. This includes **netDrv**, **nfs**, or local file devices. It does not include sockets.

EXAMPLE

```
fdX = open ("/devX/objFile", O_RDONLY);
loadModule (fdX, LOAD_ALL_SYMBOLS);
close (fdX);
```

This code fragment would load the object file "objFile" located on device "/devX/" into memory which would be allocated from the system memory pool (heap).

All external and static definitions from the file would be added to the system symbol table.

This could also have been accomplished from the shell, by typing:

```
-> ld (1) </devX/objFile
```

INCLUDE FILE

loadLib.h

ERRNOS

Routines from this library can return the following module loading specific errors:

S_loadLib_ROUTINE_NOT_INSTALLED

The routine used to load the module is not available.

S_loadLib_ILLEGAL_FLAGS_COMBINATION

The combination of specified load flags is invalid because the flags are mutually exclusive.

S_loadLib_INVALID_LOAD_FLAG

The specified load flag is invalid or does not exist.

S_loadLib_UNDEFINED_REFERENCES

There were undefined references to symbols when loading the module.

S_loadLib_INVALID_ARGUMENT

The argument specified is invalid.

S_loadLib_SDA_NOT_SUPPORTED

The kernel module loader does not support modules containing Small Data Area relocations (this errno applies to the PowerPC architecture only).

S_loadLib_MISSING_SYMBOL_TABLE

No symbol table can be found in the module (a symbol table is mandatory when loading a relocatable module).

Note that other errnos, not listed here, may come from libraries internally used by the loadable module management library.

SEE ALSO

usrLib, symLib, memLib, unldLib, moduleLib

logLib

NAME	logLib – message logging library
ROUTINES	logInit() – initialize message logging library logMsg() – log a formatted error message logFdSet() – set the primary logging file descriptor logFdAdd() – add a logging file descriptor logFdDelete() – delete a logging file descriptor logTask() – message-logging support task
DESCRIPTION	<p>This library handles message logging. It is usually used to display error messages on the system console, but such messages can also be sent to a disk file or printer.</p> <p>The routines logMsg() and logTask() are the basic components of the logging system. The logMsg() routine has the same calling sequence as printf(), but instead of formatting and outputting the message directly, it sends the format string and arguments to a message queue. The task logTask() waits for messages on this message queue. It formats each message according to the format string and arguments in the message, prepends the ID of the sender, and writes it on one or more file descriptors that have been specified as logging output streams (by logInit() or subsequently set by logFdSet() or logFdAdd()).</p>

USE IN INTERRUPT SERVICE ROUTINES

Because **logMsg()** does not directly cause output to I/O devices, but instead simply writes to a message queue, it can be called from an interrupt service routine as well as from tasks. Normal I/O, such as **printf()** output to a serial port, cannot be done from an interrupt service routine.

DEFERRED LOGGING

Print formatting is performed within the context of **logTask()**, rather than the context of the task calling **logMsg()**. Since formatting can require considerable stack space, this can reduce stack sizes for tasks that only need to do I/O for error output.

However, this also means that the arguments to **logMsg()** are not interpreted at the time of the call to **logMsg()**, but rather are interpreted at some later time by **logTask()**. This means that the arguments to **logMsg()** should not be pointers to volatile entities. For example, pointers to dynamic or changing strings and buffers should not be passed as arguments to be formatted. Thus the following would not give the desired results:

```
doLog (which)
{
    char string [100];

    strcpy (string, which ? "hello" : "goodbye");
    ...
    logMsg (string);
}
```

By the time **logTask()** formats the message, the stack frame of the caller may no longer exist and the pointer *string* may no longer be valid. On the other hand, the following is correct since the string pointer passed to the **logTask()** always points to a static string:

```
doLog (which)
{
    char *string;

    string = which ? "hello" : "goodbye";
    ...
    logMsg (string);
}
```

CONFIGURATION	To use the message logging library, configure VxWorks with the INCLUDE_LOGGING component.
INITIALIZATION	To initialize the message logging facilities, the routine loginInit() must be called before calling any other routine in this module. This is done automatically when the INCLUDE_LOGGING component is included.
INCLUDE FILES	logLib.h
SEE ALSO	msgQLib , the VxWorks programmer guides.

loginLib

NAME	loginLib – user login/password subroutine library
ROUTINES	loginInit() – initialize the login table loginUserAdd() – add a user to the login table loginUserDelete() – delete a user entry from the login table loginUserVerify() – verify a user name and password in the login table loginUserShow() – display the user login table loginPrompt() – display a login prompt and validate a user entry loginStringSet() – change the login string loginEncryptInstall() – install an encryption routine loginDefaultEncrypt() – default password encryption routine
DESCRIPTION	This library provides a login/password facility for network access to the VxWorks shell. When installed, it requires a user name and password match to gain access to the VxWorks shell from rlogin or telnet. Therefore VxWorks can be used in secure environments where access must be restricted.

Routines are provided to prompt for the user name and password, and verify the response by looking up the name/password pair in a login user table. This table contains a list of user names and encrypted passwords that will be allowed to log in to the VxWorks shell remotely. Routines are provided to add, delete, and access the login user table. The list of user names can be displayed with **loginUserShow()**.

INSTALLATION

The login security feature is initialized by the root task, **usrRoot()**, in **usrConfig.c**, if the configuration macro **INCLUDE_SECURITY** is defined. Defining this macro also adds a single default user to the login table. The default user and password are defined as **LOGIN_USER_NAME** and **LOGIN_PASSWORD**. These can be set to any desired name and password. More users can be added by making additional calls to **loginUserAdd()**. If **INCLUDE_SECURITY** is not defined, access to VxWorks will not be restricted and secure.

The name/password pairs are added to the table by calling **loginUserAdd()**, which takes the name and an encrypted password as arguments. The VxWorks host tool **vxencrypt** is used to generate the encrypted form of a password. For example, to add a user name of "fred" and password of "flintstone", first run **vxencrypt** on the host to find the encryption of "flintstone" as follows:

```
% vxencrypt
please enter password: flintstone
encrypted password is ScebRez9c
```

Then invoke the routine **loginUserAdd()** in VxWorks:

```
loginUserAdd ("fred", "ScebRez9c");
```

This can be done from the shell, a start-up script, or application code.

LOGGING IN

When the login security facility is installed, every attempt to rlogin or telnet to the VxWorks shell will first prompt for a user name and password.

```
% rlogin target

VxWorks login: fred
Password: flintstone

->
```

The delay in prompting between unsuccessful logins is increased linearly with the number of attempts, in order to slow down password-guessing programs.

ENCRYPTION ALGORITHM

This library provides a simple default encryption routine, **loginDefaultEncrypt()**. This algorithm requires that passwords be at least 8 characters and no more than 40 characters.

The routine **loginEncryptInstall()** allows a user-specified encryption function to be used instead of the default.

INCLUDE FILES

loginLib.h

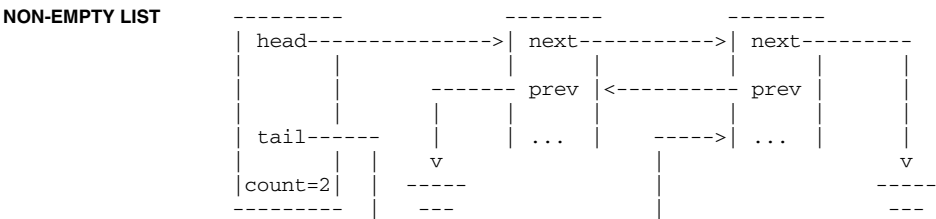
SEE ALSO **shellLib**, **vxencrypt**, the VxWorks programmer's guides

lstLib

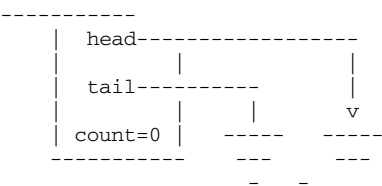
NAME	lstLib – doubly linked list subroutine library
ROUTINES	lstInit() – initialize a list descriptor lstAdd() – add a node to the end of a list lstConcat() – concatenate two lists lstCount() – report the number of nodes in a list lstDelete() – delete a specified node from a list lstExtract() – extract a sublist from a list lstFirst() – find first node in list lstGet() – delete and return the first node from a list lstInsert() – insert a node in a list after a specified node lstLast() – find the last node in a list lstNext() – find the next node in a list lstNth() – find the Nth node in a list lstPrevious() – find the previous node in a list lstNStep() – find a list node <i>nStep</i> steps away from a specified node lstFind() – find a node in a list lstFree() – free up a list

DESCRIPTION This subroutine library supports the creation and maintenance of a doubly linked list. The user supplies a list descriptor (type LIST) that will contain pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list can be any user-defined structure, but they must reserve space for two pointers as their first elements. Both the forward and backward chains are terminated with a **NULL** pointer.

The linked-list library simply manipulates the linked-list data structures; no kernel functions are invoked. In particular, linked lists by themselves provide no task synchronization or mutual exclusion. If multiple tasks will access a single linked list, that list must be guarded with some mutual-exclusion mechanism (e.g., a mutual-exclusion semaphore).



EMPTY LIST



INCLUDE FILES **lstLib.h**

m85xxCCSR

NAME	m85xxCCSR – VxBus driver for PowerPC 85xx CCSR resource allocation
ROUTINES	m85xxCCSRRegister() – register m85xxLAWBAR driver
DESCRIPTION	This is the VxBus driver for PowerPC 85xx CCSR resource management. The PowerPC 85xx processors provide address management by use of up to seven LAWBAR mapping registers. This driver allows downstream devices to allocate LAWBAR/LAWAR register sets.
INCLUDE FILES	none

mathALib

NAME	mathALib – C interface library to high-level math functions
ROUTINES	acosf() – compute an arc cosine (ANSI) atanf() – compute an arc tangent (ANSI) atan2f() – compute the arc tangent of y/x (ANSI) asinf() – compute an arc sine (ANSI) cbrt() – compute a cube root cbrtf() – compute a cube root ceilf() – compute the smallest integer greater than or equal to a specified value (ANSI) cosf() – compute a cosine (ANSI)

coshf() – compute a hyperbolic cosine (ANSI)
expf() – compute an exponential value (ANSI)
fabsf() – compute an absolute value (ANSI)
floorf() – compute the largest integer less than or equal to a specified value (ANSI)
fmodf() – compute the remainder of x/y (ANSI)
infinity() – return a very large double
infinityf() – return a very large float
rintf() – convert a double-precision value to an integer
rintf() – convert a single-precision value to an integer
round() – round a number to the nearest integer
roundf() – round a number to the nearest integer
logf() – compute a natural logarithm (ANSI)
log2() – compute a base-2 logarithm
log2f() – compute a base-2 logarithm
log10f() – compute a base-10 logarithm (ANSI)
powf() – compute the value of a number raised to a specified power (ANSI)
round() – round a number to the nearest integer
roundf() – round a number to the nearest integer
sinf() – compute a sine (ANSI)
sinhf() – compute a hyperbolic sine (ANSI)
sincos() – compute both a sine and cosine
sincosf() – compute both a sine and cosine
sqrtf() – compute a non-negative square root (ANSI)
tanf() – compute a tangent (ANSI)
tanhf() – compute a hyperbolic tangent (ANSI)
trunc() – truncate to integer
truncf() – truncate to integer

ADDITIONAL ROUTINES

acos() - compute an arc cosine (ANSI)
asin() - compute an arc sine (ANSI)
atan() - compute an arc tangent (ANSI)
atan2() - compute the arc tangent of y/x (ANSI)
ceil() - compute the smallest integer greater than or equal to a specified value (ANSI)
cos() - compute a cosine (ANSI)
cosh() - compute a hyperbolic cosine (ANSI)
exp() - compute an exponential value (ANSI)
fabs() - compute an absolute value (ANSI)
floor() - compute the largest integer less than or equal to a specified value (ANSI)
fmod() - compute the remainder of x/y (ANSI)
log() - compute a natural logarithm (ANSI)
log10() - compute a base-10 logarithm (ANSI)
pow() - compute the value of a number raised to a specified power (ANSI)
sin() - compute a sine (ANSI)
sinh() - compute a hyperbolic sine (ANSI)

sqrt() - compute a non-negative square root (ANSI)

tan() - compute a tangent (ANSI)

tanh() - compute a hyperbolic tangent (ANSI)

This reference entry describes the C interface to high-level floating-point math functions, which can use either a hardware floating-point unit or a software floating-point emulation library. The appropriate routine is called based on whether **mathHardInit()** or **mathSoftInit()** or both have been called to initialize the interface.

All angle-related parameters are expressed in radians. All functions in this library with names corresponding to ANSI C specifications are ANSI compatible.

WARNING	Not all functions in this library are available on all architectures. The architecture-specific supplements for VxWorks list any math functions that are not available.
INCLUDE FILES	math.h
SEE ALSO	ansiMath , fppLib , floatLib , mathHardLib , mathSoftLib , Kernighan & Ritchie:, <i>The C Programming Language, 2nd Edition</i> , VxWorks Architecture Supplements

memDrv

NAME	memDrv – pseudo memory device driver
ROUTINES	memDrv() – install a memory driver memDevCreate() – create a memory device memDevCreateDir() – create a memory device for multiple files memDevDelete() – delete a memory device
DESCRIPTION	<p>This driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.</p> <p>Additionally, it can be used to build some files into a VxWorks binary image (having first converted them to data arrays in C source files, using a utility such as memdrvbuild), and then mount them in the filesystem; this is a simple way of delivering some non-changing files with VxWorks. For example, a system with an integrated web server may use this technique to build some HTML and associated content files into VxWorks.</p> <p>memDrv can be used to simply provide a high-level method of reading and writing bytes in absolute memory locations through I/O calls. It can also be used to implement a simple,</p>

essentially read-only filesystem (existing files can be rewritten within their existing sizes); directory searches and a limited set of IOCTL calls (including **stat()**) are supported.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Four routines, however, can be called directly: **memDrv()** to initialize the driver, **memDevCreate()** and **memDevCreateDir()** to create devices, and **memDevDelete()** to delete devices.

Before using the driver, it must be initialized by calling **memDrv()**. This routine should be called only once, before any reads, writes, or **memDevCreate()** calls. It may be called from **usrRoot()** in **usrConfig.c** or at some later point.

IOCTL FUNCTIONS

The dosFs file system supports the following **ioctl()** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions may be any file descriptor which is opened to a file or directory on the volume or to the volume itself.

FIOGETFL

Copies to *flags* the open mode flags of the file (**O_RDONLY**, **O_WRONLY**, **O_RDWR**):

```
int flags;
status = ioctl (fd, FIOGETFL, &flags);
```

FIOSEEK

Sets the current byte offset in the file to the position specified by *newOffset*:

```
status = ioctl (fd, FIOSEEK, newOffset);
```

The FIOSEEK offset is always relative to the beginning of the file. The offset, if any, given at open time by using pseudo-file name is overridden.

FIOWHERE

Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

```
position = ioctl (fd, FIOWHERE, 0);
```

FIONREAD

Copies to *unreadCount* the number of unread bytes in the file:

```
int unreadCount;
status = ioctl (fd, FIONREAD, &unreadCount);
```

FIOREADDIR

Reads the next directory entry. The argument *dirStruct* is a DIR directory descriptor. Normally, the **readdir()** routine is used to read a directory, rather than using the FIOREADDIR function directly. See **dirLib**.

```
DIR dirStruct;
fd = open ("directory", O_RDONLY);
status = ioctl (fd, FIOREADDIR, &dirStruct);
```

FIOFSTATGET

Gets file status information (directory entry data). The argument *statStruct* is a pointer to a stat structure that is filled with data describing the specified file. File inode numbers, user and group IDs, and times are not supported (returned as 0).

Normally, the **stat()** or **fstat()** routine is used to obtain file information, rather than using the FIOFSTATGET function directly. See **dirLib**.

```
struct stat statStruct;
fd = open ("file", O_RDONLY);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```

Any other **ioctl()** function codes will return error status.

CONFIGURATION	To use the pseudo memory device driver, configure VxWorks with the INCLUDE_MEMDRV component.
INCLUDE FILES	memDrv.h
SEE ALSO	the VxWorks programmer guides.

memEdrLib

NAME	memEdrLib – memory manager error detection and reporting library
ROUTINES	memEdrFreeQueueFlush() – flush the free queue memEdrBlockMark() – mark or unmark selected blocks
DESCRIPTION	This library provides a runtime error detection and debugging tool for memory manager libraries (memPartLib and memLib). It operates by maintaining a database of blocks allocated, freed and reallocated by the memory manager and by validating memory manager operations using the database.
CONFIGURATION	<p>In the kernel, this library is enabled by including the INCLUDE_MEM_EDR component. This also requires that the ED&R logging facility (INCLUDE_EDR_ERRLOG) is also enabled. Optionally, for compiler-assisted pointer validation also include INCLUDE_MEM_EDR_RTC.</p> <p>The following component configuration parameters can also be changed:</p> <p>MEDR_EXTENDED_ENABLE Set to TRUE to enable logging trace information for each allocated block. Default setting is FALSE.</p>

MEDR_FILL_FREE_ENABLE

Set to **TRUE** to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. Default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN

Length of the free queue. Queuing is disabled when this parameter is 0. Default setting is 64.

MEDR_BLOCK_GUARD_ENABLE

Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, underruns, and some heap memory corruption, but results in a per-block allocation overhead of 16 bytes. Default setting is **FALSE**.

MEDR_POOL_SIZE

Set the size of the memory pool used to maintain the memory block database. Default setting is 1MBytes in the kernel, and 64k in RTPs. The database uses 32 bytes per memory block without extended information (call stack trace) enabled, and 64 bytes per block with extended information enabled.

When this library is enabled, the following types of memory manager errors are detected:

- allocating already allocated memory (possible heap corruption)
- allocating with invalid memory partition ID
- freeing a dangling pointer
- freeing non-allocated memory
- freeing a partial block
- freeing global memory
- freeing with invalid partition ID

The errors are logged via the ED&R facility, which should to be included in the kernel configuration. The logs can be viewed with the ED&R show routines and show commands.

FREE QUEUE AND FREE PATTERN

Freed and reallocated blocks are stored in a queue. The queue allows detection of stall pointer dereferencing in freed and re-allocated blocks. The length of the queue is set by **MEDR_FREE_QUEUE_LEN**.

When the **MEDR_FILL_FREE_ENABLE** option is enabled, queued blocks are filled with a special pattern. When the block is removed from the queue, the pattern is matched to detect memory write operations with stale pointer.

When a partition has insufficient memory to satisfy an allocation, the free queue is automatically flushed for that partition. This way the queueing does not cause allocations to fail with insufficient memory while there are blocks in the free queue.

Blocks being freed by RTPs while executing system calls are not queued. This is because an RTP's memory context may not include the mapping needed to access partitions created using memory from a shared data region.

COMPILER INSTRUMENTATION

Code compiled by the Wind River Compiler with RTEC instrumentation enabled (`-Xrtc=code` option) provides automatic pointer reference and pointer arithmetic validation.

In the kernel, this feature can be enabled with the `INCLUDE_MEM_EDR_RTC` component (in addition to the already mentioned `INCLUDE_MEM_EDR` and ED&R facility components).

Dynamically downloaded kernel modules compiled with RTEC instrumentation must be processed as C++ modules in order to enable the compiler-generated constructors and destructors.

The errors are logged via the ED&R facility, which should to be included in the kernel configuration. The logs can be viewed with the ED&R show routines and show commands.

For more information about the RTEC compiler option consult the Wind River Compiler documentation.

Note: the stack overflow check option (`-Xrtc=0x04`) is not supported with this library. Code executed in ISR or kernel context is excluded from compiler instrumentation checks.

CAVEATS

Realloc does not attempt to resize a block. Instead, it will always allocate a new block and enqueue the old block into the free queue. This method enables detection of invalid references to reallocated blocks.

Realloc with size 0 will return a pointer to a block of size 0. This feature coupled with compiler pointer validation instrumentation aids in detecting dereferencing pointers obtained by realloc with size 0.

In order to aid detection of unintended free and realloc operation on invalid pointers, memory partitions should not be created in a task's stack when this library is enabled. Although it is possible to create such memory partitions, it is not a recommended practice; this library will flag it as an error when an allocated block is within a task's own stack.

Memory partition information is recorded in the database for each partition created. This information is kept even after the memory partition is deleted, so that unintended operations with a deleted partition can be detected.

INCLUDE FILES none

SEE ALSO `memEdrShow`, `memEdrRtpShow`, `edrLib`, `memLib`, `memPartLib`

memEdrRtpShow

NAME	memEdrRtpShow – memory error detection show routines for RTPs
ROUTINES	memEdrRtpPartShow() – show partition information of an RTP memEdrRtpBlockShow() – print memory block information of an RTP memEdrRtpBlockMark() – mark or unmark selected allocated blocks in an RTP
DESCRIPTION	This module provides the show routines of the memory manager instrumentation and error detection library for RTPs. To use these libraries, configure VxWorks with INCLUDE_MEM_EDR_RTP and INCLUDE_MEM_EDR_RTP_SHOW . In addition, set the MEDR_SHOW_ENABLE environment variable to TRUE .
INCLUDE FILES	none
SEE ALSO	memEdrLib

memEdrShow

NAME	memEdrShow – memory error detection show routines
ROUTINES	memEdrPartShow() – show partition information in the kernel memEdrBlockShow() – print memory block information
DESCRIPTION	This module provides show routines for the memory manager instrumentation and error detection library.
CONFIGURATION	To use the memory error detection show routines, configure VxWorks with the INCLUDE_MEM_EDR and INCLUDE_MEM_EDR_SHOW components.
INCLUDE FILES	none
SEE ALSO	memEdrLib

memInfo

NAME	memInfo – memory partition info routines
------	---

ROUTINES	memPartInfoGet() – get partition information memPartFindMax() – find the size of the largest available free block memInfoGet() – get heap information memFindMax() – find the largest free block in the system memory partition (kernel heap)
DESCRIPTION	This library provides routines for obtaining information about a memory partition or the kernel heap. It is included in a kernel image configuration via the INCLUDE_MEM_MGR_INFO components.
INCLUDE FILES	memLib.h
SEE ALSO	memPartLib , memLib , memShow

memLib

NAME	memLib – full-featured memory partition manager
ROUTINES	memPartOptionsSet() – set the options for a memory partition memPartOptionsGet() – get the options of a memory partition memalign() – allocate aligned memory from system memory partition (kernel heap) valloc() – allocate memory on a page boundary from the kernel heap memPartRealloc() – reallocate a block of memory in a specified partition memOptionsSet() – set the options for the system memory partition (kernel heap) memOptionsGet() – get the options of the system memory partition (kernel heap)
DESCRIPTION	<p>This library provides full-featured facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library is an extension of memPartLib and provides enhanced memory management features, including error handling, aligned allocation, and ANSI allocation routines. For more information about the core memory partition management facility, see the manual entry for memPartLib.</p> <p>The system memory partition, which can also be referred to as the kernel heap, is automatically created when the kernel is initialized.</p> <p>The memalign() routine is provided for allocating memory aligned to a specified boundary.</p>
CONFIGURATION	<p>To use the memory partition manager, configure VxWorks with the INCLUDE_MEM_MGR_FULL component.</p> <p>Various debug options can be selected for each partition using memPartOptionsSet() and memOptionsSet(). Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. The following error-handling options can be individually selected:</p>

MEM_ALLOC_ERROR_EDR_FATAL_FLAG

Inject a fatal ED&R event when there is an error in allocating memory. This option takes precedence over the **MEM_ALLOC_ERROR_EDR_WARN_FLAG** and **MEM_ALLOC_ERROR_SUSPEND_FLAG** options.

MEM_ALLOC_ERROR_EDR_WARN_FLAG

Inject an ED&R warning when there is an error in allocating memory.

MEM_ALLOC_ERROR_LOG_FLAG

Log a message when there is an error in allocating memory.

MEM_ALLOC_ERROR_SUSPEND_FLAG

Suspend the task when there is an error in allocating memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended). This option has been deprecated (available for backward compatibility only).

MEM_BLOCK_ERROR_EDR_FATAL_FLAG

Inject a fatal ED&R event when there is an error in freeing or reallocating memory. This option takes precedence over the **MEM_BLOCK_ERROR_EDR_WARN_FLAG** and **MEM_BLOCK_ERROR_SUSPEND_FLAG** options.

MEM_BLOCK_ERROR_EDR_WARN_FLAG

Inject a non-fatal ED&R event when there is an error in freeing or reallocating memory.

MEM_BLOCK_ERROR_LOG_FLAG

Log a message when there is an error in freeing memory.

MEM_BLOCK_ERROR_SUSPEND_FLAG

Suspend the task when there is an error in freeing memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended). This option will be deprecated in future releases.

When the following option is specified to check every block freed to the partition, **memPartFree()** and **free()** in **memPartLib** run consistency checks of various pointers and values in the header of the block being freed. If this flag is not specified, no check will be performed when memory is freed.

MEM_BLOCK_CHECK

Check each block freed.

Setting any of the **MEM_BLOCK_ERROR_** options automatically sets **MEM_BLOCK_CHECK**.

The options of a partition are initialized to the value of the **MEM_PART_DEFAULT_OPTIONS** configuration parameter, which defaults to the following flags being enabled:

MEM_ALLOC_ERROR_LOG_FLAG
MEM_ALLOC_ERROR_EDR_WARN_FLAG
MEM_BLOCK_CHECK
MEM_BLOCK_ERROR_LOG_FLAG
MEM_BLOCK_ERROR_EDR_WARN_FLAG
MEM_BLOCK_ERROR_SUSPEND_FLAG

When setting options for a partition with **memPartOptionsSet()** or **memOptionsSet()**, use the logical OR operator between each specified option to construct the *options* parameter. For example:

```
memPartOptionsSet (myPartId, MEM_ALLOC_ERROR_LOG_FLAG |
                    MEM_BLOCK_CHECK |
                    MEM_BLOCK_ERROR_LOG_FLAG) ;
```

In the case when multiple options are set so that one option takes precedence over the other, then the preceeded options may not have their expected effect. For example, if the **MEM_BLOCK_ERROR_EDR_FATAL_FLAG** flag results in a task being stopped by the ED&R fatal policy handler, then the **MEM_BLOCK_ERROR_SUSPEND_FLAG** flag has no effect (a task cannot be stopped and suspended at the same time).

KERNEL VERSUS RTP HEAP ALLOCATOR

Memory allocated in user code running in an RTP is managed by the RTP heap allocator, independent from the kernel heap. Each RTP has its own heap. By default, this service is provided by the user version of **memLib** and **memPartLib**. These libraries can be replaced with other third party or user provided allocators. For more information see also the documentation for the user **memLib** and **memPartLib**.

INCLUDE FILES **memLib.h**

SEE ALSO **memPartLib, smMemLib**

memPartLib

NAME **memPartLib** – core memory partition manager

ROUTINES

- memPartCreate()** – create a memory partition
- memPartDelete()** – delete a partition and free associated memory
- memPartAddToPool()** – add memory to a memory partition
- memPartAlignedAlloc()** – allocate aligned memory from a partition
- memPartAlloc()** – allocate a block of memory from a partition
- memPartFree()** – free a block of memory in a partition
- memAddToPool()** – add memory to the system memory partition
- malloc()** – allocate a block of memory from the system memory partition (ANSI)
- calloc()** – allocate space for an array (ANSI)
- realloc()** – reallocate a block of memory (ANSI)
- free()** – free a block of memory from the system memory partition (ANSI)
- cfree()** – free a block of memory from the system memory partition (kernel heap)

DESCRIPTION This library provides core facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library was designed to provide a compact implementation; full-featured functionality is available with **memLib**, which provides enhanced memory management features built as an extension of **memPartLib**. (For more information about enhanced memory partition management options, see the manual entry for **memLib**.) This library consists of two sets of routines. The first set, **memPart...()**, comprises a general facility for the creation and management of memory partitions, and for the allocation and deallocation of blocks from those partitions. The second set provides a traditional ANSI-compatible **malloc()**/**free()** interface to the system memory partition.

The system memory partition, which can also be referred to as the kernel heap, is automatically created when the kernel is initialized.

The allocation of memory, using **malloc()** in the typical case and **memPartAlloc()** for a specific memory partition, is done with a best-fit algorithm. Adjacent blocks of memory are coalesced when they are freed with **memPartFree()** and **free()**. There is also a routine provided for allocating memory aligned to a specified boundary from a specific memory partition, **memPartAlignedAlloc()**.

This library includes three ANSI-compatible routines: **calloc()** allocates a block of memory for an array; **realloc()** changes the size of a specified block of memory; and **cfree()** returns to the free memory pool a block of memory that was previously allocated with **calloc()**.

CONFIGURATION This library is always included in VxWorks.

CAVEATS Architectures have various alignment constraints. To provide optimal performance, **malloc()** returns a pointer to a buffer having the appropriate alignment for the architecture in use. The portion of the allocated buffer reserved for system bookkeeping, known as the overhead, may vary depending on the architecture. The following table lists the default alignment and overhead size of free and allocated memory blocks for various architectures.

Architecture	Boundary	Overhead
ARM	4	16
COLDFIRE	4	16
I86	4	16
M68K	4	16
MCORE	8	16
MIPS	16	16
PPC (*)	8-16	16
SH	4	16
SIMLINUX	8	16
SIMNT	8	16
SIMSOLARIS	8	16
SPARC	8	16

(*) On PowerPC, the boundary and allocated block overhead values are 16 bytes for system based on the PPC604 CPU type (including ALTIVEC). For all other PowerPC CPU types

(PPC403, PPC405, PPC440, PPC860, PPC603, etc...), the boundary for allocated blocks is 8 bytes.

The partition's free blocks are organized into doubly linked lists. Each list contains only free blocks of the same size. The head of these doubly linked lists are organized in an AVL tree. The memory for the AVL tree's nodes is carved out from the partition space itself, whenever new AVL nodes need to be created. This occurs only if the fragmentation of the partition increases; to be more exact, it happens only if a free memory block is created whose size does not have a doubly linked list yet. This amount of memory carved out from the partition space for bookkeeping purposes is reported by **memPartShow()** and **memShow()** as the amount of "internal" memory allocated in the partition.

INCLUDE FILES **memPartLib.h, stdlib.h**

SEE ALSO **memLib, smMemLib**

memShow

NAME **memShow** – memory show routines

ROUTINES **memShowInit()** – initialize the memory partition show facility
memShow() – show blocks and statistics for the current heap partition
memPartShow() – show blocks and statistics for a given memory partition

DESCRIPTION This library contains memory partition information display routines. To use this facility, it must first be installed using **memShowInit()**, which is called automatically when the memory partition show facility is configured into VxWorks. To configure the memory partition show facility into VxWorks, include the **INCLUDE_MEM_SHOW** component.

INCLUDE FILES none

SEE ALSO **memLib, memPartLib**, the VxWorks programmer guides, the IDE and host tools guides.

miiBus

NAME **miiBus** – MII bus controller and API library

ROUTINES **miiBusRegister()** – register with the vxBus subsystem
miiBusListAdd() – Add a PHY to the MII monitor list

miiBusListDel() – Remove a PHY to the MII monitor list
miiBusGet() – get the miiBus that goes with a given VxBus instance
miiBusCreate() – create an miiBus attached to a parent bridge
miiBusDelete() – delete an miiBus and all its child devices
miiBusRead() – read a PHY register
miiBusWrite() – write value to a PHY register
miiBusMediaUpdate() – invoke a PHY's parent's media update callback
miiBusModeGet() – get the current media mode and link status
miiBusModeSet() – set the current media mode
miiBusMediaListGet() – obtain a pointer to the bus's media list
miiBusMediaAdd() – add an entry to an miiBus's media list
miiBusMediaDel() – delete an entry to an miiBus's media list
miiBusMediaDefaultSet() – set the default media for an miiBus

DESCRIPTION

This module implements a VxBus bus controller for managing MII-compliant ethernet physical layer interfaces (PHYs). Up to 32 PHYs can be connected to a single MII management bus. In the most common case, an ethernet controller will have a single PHY attached to it, and that PHY's MDIO pins will be connected to the ethernet controller and accessed through MDIO registers in the controller's register space. This module is designed to allow an ethernet controller's MDIO bus to be abstracted as a VxBus bus, and for PHYs to be probed and attached as VxBus instances. The main benefits to this are that drivers can be provided for specific PHY chips where necessary (although PHYs are meant to have a generic management interface, it's often necessary to configure vendor-specific registers to get some chips to work correctly), and save driver developers from having to duplicate PHY management code over and over in each ethernet driver.

Each ethernet controller that uses an MII-based (or MII-like) PHY must have a logical MII bus. The bus is created using the **miiBusCreate()** function, with the ethernet controller's VxBus instance as a bridge. This will create the bus instance, and probe for PHYs attached to the bus. Probing is accomplished using the **miiBusRead()** and **miiBusWrite()** functions. These functions depend on the ethernet driver providing two VxBus methods: **miiRead** and **miiWrite**. These methods should be implemented in the driver, and provide a way to read a specific PHY register at a given PHY address. Once a bus is created, the **miiBusGet()** function can be used to obtain a pointer to its VxBus instance. The ethernet driver should save this pointer and use it to access the bus later on.

The ethernet driver should also export an **miiMediaUpdate** method. This is a callback which may be invoked by PHY drivers as the result of media change events, such as connecting or disconnecting of the network cable. This callback should be used to configure the controller to match the PHY when the link state changes. For example, some ethernet MACs must be explicitly set for full or half duplex mode to match the duplex setting of the PHY. Consequently, the MAC must be alerted if the duplex setting of the link changes.

INCLUDE FILES

none

mmanPxLib

NAME	mmanPxLib – memory management library (POSIX)
ROUTINES	mlockall() – lock all pages used by a process into memory (POSIX) munlockall() – unlock all pages used by a process (POSIX) mlock() – lock specified pages into memory (POSIX) munlock() – unlock specified pages (POSIX)
DESCRIPTION	This library contains POSIX interfaces designed to lock and unlock memory pages, i.e., to control whether those pages may be swapped to secondary storage. Since VxWorks does not use swapping (all pages are always kept in memory), these routines have no real effect and simply return 0 (OK).
INCLUDE FILES	sys/mman.h
SEE ALSO	POSIX 1003.1b document

mmanShow

NAME	mmanShow – mmap manager show library
ROUTINES	mmapShow() – show information about memory mapped objects in the system
DESCRIPTION	<p>This library provides routines to display information about memory mapped objects (files and mapped shard memory objects). These objects are mapped in a process' address space with mmap().</p> <p>This library is automatically initialized whenever the component INCLUDE_MAPPED_FILES_SHOW is added.</p>
INCLUDE FILES	n/a
SEE ALSO	application_mmanLib

mmuMapLib

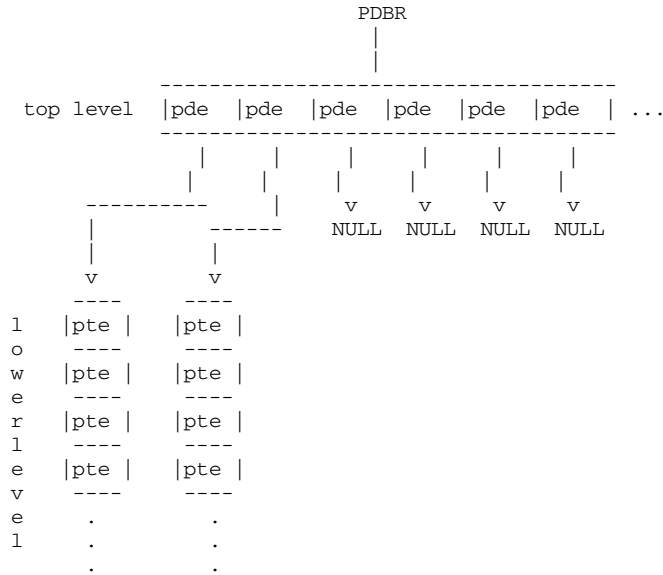
NAME	mmuMapLib – MMU mapping library for ARM Ltd. processors
------	--

ROUTINES	mmuVirtToPhys() – translate a virtual address to a physical address (ARM) mmuPhysToVirt() – translate a physical address to a virtual address (ARM)
DESCRIPTION	This library provides additional MMU support routines. These are present in a separate module from mmuLib.c , so that these routines can be used without including all the code in that object module.
INCLUDE FILES	none

mmuPro32Lib

NAME	mmuPro32Lib – MMU library for Pentium II
ROUTINES	mmuPro32LibInit() – initialize module mmuPro32Page0UnMap() – unmap the page zero for NULL pointer detection
DESCRIPTION	<p>mmuLib.c provides the architecture dependent routines that directly control the memory management unit. It provides routines that are called by the higher level architecture independent routines in vmLib.c:</p> <p>mmuLibInit - initialize module mmuTransTblCreate - create a new translation table mmuTransTblDelete - delete a translation table. mmuEnable - turn mmu on or off mmuStateSet - set state of virtual memory page mmuStateGet - get state of virtual memory page mmuPageMap - map physical memory page to virtual memory page mmuPageUnMap - unmap a physical page. mmuTranslate - translate a virtual address to a physical address mmuCurrentSet - change active translation table mmuTransTblUnion - merge two translation tables mmuTransTblMask - subtract one translation table from another. mmuAttrTranslate - translate special VM attributes to Intel bits. mmuBufferWrite - writes to any mapped memory page w/o changing attributes. mmuPageSizeGet - return the page size</p> <p>Applications using the mmu will never call these routines directly; the visible interface is supported in vmLib.c.</p> <p>mmuPro32Lib supports the creation and maintenance of multiple translation tables. New translation tables are created with a call to mmuTransTblCreate(). The translation table is initialized by allocating a 4KB page to serve as a Page Directory Table for the new context. (The first time mmuTransTblCreate is called, it initializes the kernel's virtual memory context. Subsequent calls initialize other supervisor or user contexts.) After the table is created and initialized, pages can be mapped using mmuPageMap(). Page mapping associates physical addresses with virtual addresses. The attributes of any mapped page can then be changed by calling mmuStateSet(). "Attributes" are read/write, user/supervisor, writethrough/copyback, cache on/off. A translation table is installed as the currently active table by mmuCurrentSet().</p>

The typical translation table looks like this:



The physical memory that holds these data structures is obtained from the system memory manager via `malign` to insure that the memory is page aligned and is located in a transparently mapped region (virtual address equals physical address). In order to protect the page tables themselves from being corrupted, they are made read-only after being mapped. The protection is done when the first call to `mmuCurrentSet()` is made. This point is chosen because up until then, we don't know whether the kernel's memory has been mapped.

163

used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. The PageSize bit is used only in a directory table entry. When set it indicates that the page size for that entry is 4MB and there is no level 2 page table. This bit is ignored if the page size enable bit (PSE) is not set in control register CR4.

This module supports the PentiumPro and Pentium II MMU. It is compatible with 80486 and Pentium, as long as the Global and PageSize attribute bits are not used.

INCLUDE FILES none

mmuPro36Lib

NAME **mmuPro36Lib** – MMU library for PentiumPro/2/3/4 36 bit mode

ROUTINES **mmuPro36LibInit()** – initialize module
mmuPro36Page0UnMap() – unmap the page zero for NULL pointer detection
mmuPro36PageMap() – map 36bit physical memory page to virtual memory page
mmuPro36Translate() – translate a virtual address to a 36bit physical address

DESCRIPTION **mmuPro36Lib.c** provides the architecture dependent routines that directly control the memory management unit. It provides routines that are called by the higher level architecture independent routines in **vmLib.c**:

mmuLibInit - initialize module **mmuTransTblCreate** - create a new translation table
mmuTransTblDelete - delete a translation table. **mmuEnable** - turn mmu on or off
mmuStateSet - set state of virtual memory page **mmuStateGet** - get state of virtual memory page
mmuPageMap - map physical memory page to virtual memory page
mmuPageUnMap - unmap a physical page. **mmuTranslate** - translate a virtual address to a physical address
mmuCurrentSet - change active translation table **mmuTransTblUnion** - merge two translation tables
mmuTransTblMask - subtract one translation table from another. **mmuAttrTranslate** - translate special VM attributes to Intel bits. **mmuBufferWrite** - writes to any mapped memory page w/o changing attributes. **mmuPageSizeGet** - return the page size

Applications using the mmu will never call these routines directly; the visible interface is supported in **vmLib.c**.

mmuPro36Lib supports the creation and maintenance of multiple translation tables. New translation tables are created with a call to **mmuTransTblCreate()**. The translation table is initialized by allocating a 4KB page to serve as a Page Directory Table for the new context. (The first time **mmuTransTblCreate** is called, it initializes the kernel's virtual memory context. Subsequent calls initialize other supervisor or user contexts.) After the table is created and initialized, pages can be mapped using **mmuPageMap()**. Page mapping associates physical addresses with virtual addresses. The attributes of any mapped page

1

1

1

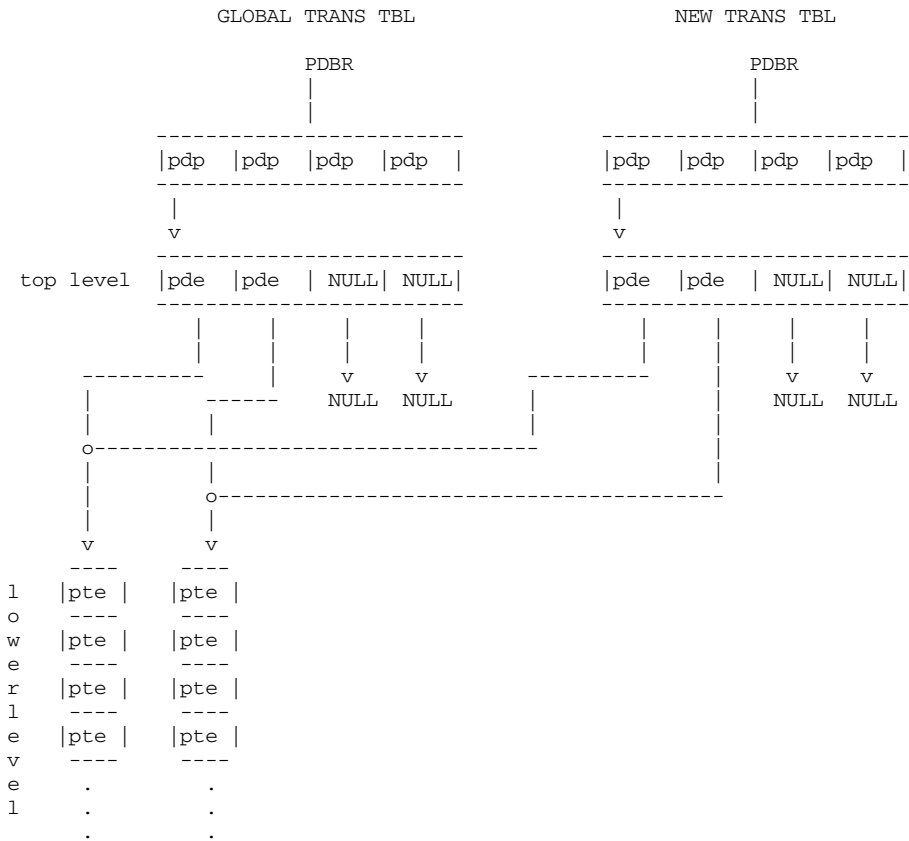
1



1

1

translation tables are created, memory for the top level array of sftd's is allocated and initialized by duplicating the pointers in mmuGlobalTransTbl's top level sftd array. Thus, the new translation table will use the global translation table's state information for portions of virtual memory that are defined as global. Here's a picture to illustrate:



Note that with this scheme, the global memory granularity is 4MB. Each time you map a section of global virtual memory, you dedicate at least 4MB of the virtual space to global virtual memory that will be shared by all virtual memory contexts.

The physical memory that holds these data structures is obtained from the system memory manager via `mmap` to insure that the memory is page aligned. We want to protect this memory from being corrupted, so we invalidate the descriptors that we set up in the global translation that correspond to the memory containing the translation table data structures. This creates a "chicken and the egg" paradox, in that the only way we can modify these data structures is through virtual memory that is now invalidated, and we can't validate it because the page descriptors for that memory are in invalidated memory (confused yet?)

So, you will notice that anywhere that page table descriptors (PTE's) are modified, we do so by locking out interrupts, momentarily disabling the MMU, accessing the memory with its physical address, enabling the MMU, and then re-enabling interrupts (see **mmuStateSet()**, for example.)

Support for two new page attribute bits are added for PentiumPro's enhanced MMU. They are Global bit (G) and Page-level write-through/back bit (PWT). Global bit indicates a global page when set. When a page is marked global and the page global enable (PGE) bit in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occurs. This bit is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Page-level write-through/back bit (PWT) controls the write-through or write-back caching policy of individual pages or page tables. When the PWT bit is set, write-through caching is enabled for the associated page or page table. When the bit is clear, write-back caching is enabled for the associated page and page table.

Following macros are used to describe these attribute bits in the physical memory descriptor table **sysPhysMemDesc[]** in **sysLib.c**.

VM_STATE_WBACK - use write-back cache policy for the page

VM_STATE_WBACK_NOT - use write-through cache policy for the page

VM_STATE_GLOBAL - set page global bit

VM_STATE_GLOBAL_NOT - not set page global bit

Support for two page size (4KB and 2MB) are added also. The linear address for 4KB pages is divided into four sections:

Page directory pointer - bits 30 through 31.

Page directory entry - bits 21 through 29.

Page table entry - Bits 12 through 20.

Page offset - Bits 0 through 11.

The linear address for 2MB pages is divided into three sections:

Page directory pointer - bits 30 through 31.

Page directory entry - Bits 21 through 29.

Page offset - Bits 0 through 20.

These two page size is configurable by **VM_PAGE_SIZE** macro in **config.h**.

INCLUDE FILES none

mmuShLib

NAME	mmuShLib – Memory Management Unit Library for Renesas SH77xx
ROUTINES	mmuShLibInit() – Initialize the SH MMU library.
DESCRIPTION	<p>The SH family of devices range between many different manufacturers resulting in a wide range of implementations of memory management units. This library contains the functions that support the SH7750 version of these devices. It provides routines that are called by the architecture independent manager (AIM). There are two layers of architecture independent routines: the lower of these is the Architecture-Independent MMU which provides page-table management, and the upper is vmLib.c or vmBaseLib.c.</p> <p>The SH MMU library is based on a page-table architecture created to handle TLB exceptions. This page-table architecture is a three level hierarchy. Level-0, the Context Table, is comprised of 256 4-byte pointers to the next level of page tables. The Level-1 page table, the Region Table, is pointed to by Level-0 and is comprised of 1024 4-byte pointers to the next level of page tables. It is indexed by the top 10 bits of the PTEH register. Level-2, the Page Table, pointed to by Level-1, contains the page-table entries used to fill the TLB. It is indexed by the second 10 bits of the PTEH register.</p> <p>The sizes of the three-level page-table architecture restrict some of the characteristics of the system. The size of the region table--limited to the number of available contexts given by the Address Space ID (if available) field of the TLB--restricts the system to only 256 separate execution contexts. The size of the L1 and L2 page tables restricts the system to a minimum page size of 4KB and a maximum page size of 1MB.</p>
INCLUDE FILES	

moduleLib

NAME	moduleLib – code module list management library
ROUTINES	<p>moduleCreate() – create and initialize a module moduleDelete() – delete module ID information moduleSegGet() – get (delete and return) the first segment from a module moduleSegFirst() – find the first segment in a module moduleSegNext() – find the next segment in a module moduleCreateHookAdd() – add a routine to be called when a module is added moduleCreateHookDelete() – delete a previously added module create hook routine moduleFindByName() – find a module by name moduleFindByNameAndPath() – find a module by filename and path</p>

moduleFindByGroup() – find a module by group number
moduleIdListGet() – get a list of loaded modules
moduleInfoGet() – get information about an object module
moduleCheck() – verify checksums on all modules loaded in the system
moduleNameGet() – get the name associated with a module ID
moduleFlagsGet() – get the flags associated with a module ID
moduleShow() – show information about loaded modules

DESCRIPTION This library is used to keep track of which object modules have been loaded into VxWorks, to maintain information about object module segments associated with each module, to track which symbols belong to which module and to maintain information about resources involved in their installation in the system's memory. Tracking modules makes it possible to list which modules are currently loaded and to unload them when they are no longer needed.

CODE MODULE AND SEGMENT DESCRIPTORS

Loading an object module requires allocating memory for its loadable sections, for the symbols it holds (depending on the exact load flags being used, see **loadModule()** for more information), and for module management information.

By convention, we refer to this collection of related items as a *code module*. So, a code module can be seen as the result of the load operation of an object file. In other words, an object module is loaded, resulting in the installation of a code module in the target's memory (see also **loadLib**).

The module management information is composed of descriptors, one for the code module itself and one for each loaded segment. The ELF sections of the file are aggregated into segments. See the **loadLib** documentation for more detail. The segment descriptors are private structures which contain various pieces of information accessible via the **moduleInfoGet()** routine.

Multiple modules with the same name are allowed (the same module may be loaded without first being unloaded) but "find" functions find the most recently created module.

NOTE In general, users will not access these routines directly, with the exception of **moduleShow()**, which displays information about currently loaded modules. Most calls to this library will be from routines in **loadLib** and **unldLib**.

INCLUDE FILES **moduleLib.h**

ERRNOS Routines from this library can return the following module list-specific errors:

S_moduleLib_BAD_CHECKSUM

The checksum on one of the registered sections is incorrect

S_moduleLib_MODULE_NOT_FOUND

The code module which is looked for can not be found in the code module lists.

S_moduleLib_MAX_MODULES_LOADED
There are too many loaded modules to perform the operation.

S_moduleLib_HOOK_NOT_FOUND
The specified hook routine can not be found in the list of registered hooks.

S_moduleLib_INVALID_SECTION_ID
The section ID passed as a parameter is invalid.

S_moduleLib_INVALID_MODULE_ID
The module ID passed as a parameter is invalid.

Note that other errors, not listed here, may come from libraries internally used by the module list management.

SEE ALSO **loadLib, unldLib, symLib, memLib**

mountd

NAME **mountd** – Mount protocol library

ROUTINES **mountdInit()** – initialize the mount daemon
 nfsExport() – specify a file system to be NFS exported
 nfsUnexport() – remove a file system from the list of exported file systems

DESCRIPTION This file implements the initialization routines for the MOUNT daemon.

The mount server is initialized by calling **mountdInit()**. This routine, will be invoked by **nfsdInit()**.

Including any of the NFS SERVER components (**INCLUDE_NFS_SERVER_ALL**, **INCLUDE_NF3_SERVER**, **INCLUDE_NFS2_SERVER**) enables the call to **nfsdInit()** during the boot process, which in turn calls **mountdInit()**, so there is normally no need to call either routine manually. **mountdInit()** spawns one task, **tMountd**, which registers as an RPC service with the portmapper. The registration is done for MOUNT V1 and/or MOUNT V3 depending on the NFS server version included. If only NFS V3 server is included then both MOUNT V1 and MOUNT V3 are registered. If only NFS V2 is included then only the MOUNT V1 is registered.

NOTE: The only routines in this library that are normally called by applications are **nfsExport()** and **nfsUnexport()**. The mount daemon is normally initialized indirectly by **nfsdInit()**.

Currently, only the **dosFsLib** file system is supported. File systems are exported with the **nfsExport()** call.

To export VxWorks file systems via NFS, you need facilities from both this library and from **nfsdLib**. To include the **nfsLib** components, use either **INCLUDE_NFS2_CLIENT** or **INCLUDE_NFS3_CLIENT** or **INCLUDE_NFS_CLIENT_ALL** and rebuild VxWorks.

Example The following example illustrates how to export an existing dosFs file system.

First, initialize the block device containing your file system.

Then assuming the dosFs system is called "/export" execute the following code on the target:

```
nfsExport ("/export", 0, FALSE, 0);           /* make available remotely */
```

This makes it available to all clients to be mounted using the client's NFS mounting command. (On UNIX systems, mounting file systems normally requires root privileges.)

VxWorks does not normally provide authentication services for NFS requests, and the DOS file system does not provide file permissions. If you need to authenticate incoming requests, see the documentation for **nfsdInit()** and **mountdInit()** for information about authorization hooks.

The following requests are accepted from clients. For details of their use, see Appendix I of RFC 1813, "NFSv3 Protocol Specification."

Procedure Name	Procedure Number
MOUNTPROC_NULL	0
MOUNTPROC_MNT	1
MOUNTPROC_DUMP	2
MOUNTPROC_UMNT	3
MOUNTPROC_UMNTALL	4
MOUNTPROC_EXPORT	5

INCLUDE FILES none

SEE ALSO dosFsLib, nfsdLib, RFC 1813

mqPxLib

NAME mqPxLib – message queue library (POSIX)

ROUTINES mqPxLibInit() – initialize the POSIX message queue library
mq_open() – open a message queue (POSIX)
mq_receive() – receive a message from a message queue (POSIX)
mq_send() – send a message to a message queue (POSIX)
mq_close() – close a message queue (POSIX)

mq_unlink() – remove a message queue (POSIX)
mq_notify() – notify a task that a message is available on a queue (POSIX)
mq_setattr() – set message queue attributes (POSIX)
mq_getattr() – get message queue attributes (POSIX)
mqPxDescObjIdGet() – returns the **OBJ_ID** associated with a **mqd_t** descriptor

DESCRIPTION This library implements the message-queue interface based on the POSIX 1003.1b standard, as an alternative to the VxWorks-specific message queue design in **msgQLib**. The POSIX message queues are accessed through names; each message queue supports multiple sending and receiving tasks.

The message queue interface imposes a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis. The value may not be changed dynamically.

This interface allows a task to be notified asynchronously of the availability of a message on the queue. The purpose of this feature is to let the task perform other functions and yet still be notified that a message has become available on the queue.

MESSAGE QUEUE DESCRIPTOR DELETION

The **mq_close()** call terminates a message queue descriptor and deallocates any associated memory. When deleting message queue descriptors, take care to avoid interfering with other tasks that are using the same descriptor. Tasks should only close message queue descriptors that the same task has opened successfully.

INCLUDE FILES **mqqueue.h**

SEE ALSO POSIX 1003.1b document, **msgQLib**

mqPxShow

NAME **mqPxShow** – POSIX message queue show

ROUTINES **mqPxShow()** – display message queue internals
mqPxShowInit() – initialize the POSIX message queue show facility

DESCRIPTION This library provides a show routine for display information on POSIX message queue objects. The information displayed is for debugging purposes and is intended to be a snapshot of the system at the time the call is made.

INCLUDE FILES **mqPxShow.h**

msgQEvLib

NAME	msgQEvLib – VxWorks events support for message queues
ROUTINES	msgQEvStart() – start the event notification process for a message queue msgQEvStop() – stop the event notification process for a message queue
DESCRIPTION	<p>This library is an extension to eventLib, the VxWorks events library. Its purpose is to support events for message queues.</p> <p>The functions in this library are used to control registration of tasks on a message queue. The routine msgQEvStart() registers a task and starts the notification process. The function msgQEvStop() un-registers the task, which stops the notification mechanism.</p> <p>When a task is registered and a message arrives on the queue, the events specified are sent to that task, on the condition that no other task is pending on that message queue. However, if a msgQReceive() is to be done afterwards to get the message, there is no guarantee that it will still be available.</p>
SMP CONSIDERATIONS	<p>Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling intCpuLock()) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.</p>
INCLUDE FILES	msgQEvLib.h
SEE ALSO	eventLib

msgQInfo

NAME	msgQInfo – message queue information routines
ROUTINES	msgQInfoGet() – get information about a message queue
DESCRIPTION	This library provides routines to show message queue statistics, such as the task queuing method, messages queued, and receivers blocked.

The routine **msgQInfoGet()** gets the information about message queues, such as the task queuing method, messages queued, and receivers blocked.

This component is required by pipe and message queue show routines. It can be configured into VxWorks using either of the following methods:

- Using the configuration header files, define **INCLUDE_MSG_Q_INFO** in **config.h**.
- Using the project facility, select **INCLUDE_MSG_Q_INFO**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**

msgQLib

NAME **msgQLib** – message queue library

ROUTINES **msgQInitialize()** – initialize a pre-allocated message queue
msgQSend() – send a message to a message queue
msgQReceive() – receive a message from a message queue
msgQNumMsgs() – get the number of messages queued to a message queue
msgQCreate() – create and initialize a message queue
msgQDelete() – delete a message queue

DESCRIPTION This library contains routines for creating and using message queues, the primary inter-task communication mechanism within a single CPU. Message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out (FIFO) order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

To provide message queue support for a system, VxWorks must be configured with the **INCLUDE_MSG_Q** component.

CREATING AND USING MESSAGE QUEUES

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

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

A task receives a message from a message queue with **msgQReceive()**. If any 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, if no buffer space is available to queue the message, the timeout specifies how many ticks to wait for space to become available. When receiving a message, the timeout specifies how many ticks to wait if no message is immediately available. The *timeout* parameter can have the special values **NO_WAIT** (0) or **WAIT_FOREVER** (-1). **NO_WAIT** means the routine returns immediately; **WAIT_FOREVER** means the routine never times out.

URGENT MESSAGES

The **msgQSend()** routine allows the priority of a message to be specified. It can be either **MSG_PRI_NORMAL** (0) or **MSG_PRI_URGENT** (1). 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.

VXWORKS EVENTS

If a task has registered with a message queue via **msgQEvStart()**, events are sent to that task when a message arrives on that message queue, on the condition that no other task is pending on the queue.

CAVEATS

There is a small difference between how pended senders and pended receivers are handled. As in previous versions of VxWorks, a pended receiver is made ready as soon as a sender sends a message.

Unlike previous versions of VxWorks, FIFO message queues allow only one pended sender can be made ready by a receive; subsequent receive operations do not unpend more senders. Instead the next sender is unpended by the previously unpended sender. This enforces the correct order of delivery of messages onto the queue. This may affect the length of time a sender spends pending for the message queue resource. Priority message queues are not affected by this restriction.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpulock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv, msgQSmLib, msgQEvLib, eventLib**

msgQOpen

NAME **msgQOpen** – extended message queue library

ROUTINES **msgQOpenInit()** – initialize the message queue open facility
msgQOpen() – open a message queue
msgQClose() – close a named message queue
msgQUnlink() – unlink a named message queue

DESCRIPTION The extended message queue library includes the APIs to open, close, and unlink message queues. Since these APIs did not exist in VxWorks 5.5, to prevent the functions from being included in the default image, they have been isolated from the general message queue library.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpulock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **msgQLib.h**

SEE ALSO **objOpen, semOpen, taskOpen, timerOpen**, the VxWorks, programmer guides

msgQShow

NAME	msgQShow – message queue show routines
ROUTINES	msgQShowInit() – initialize the message queue show facility msgQShow() – show information about a message queue
DESCRIPTION	<p>This library provides routines to show message queue statistics, such as the task queuing method, messages queued, and receivers blocked.</p> <p>The routine msgQshowInit() links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:</p> <ul style="list-style-type: none">- Using the configuration header files, define INCLUDE_SHOW_ROUTINES in config.h.- Using the project facility, select INCLUDE_MSG_Q_SHOW. <p>The msgQShow() routine displays information about message queues, such as the task queuing method, messages queued, and receivers blocked.</p>
INCLUDE FILES	msgQLib.h
SEE ALSO	pipeDrv

msgQSmLib

NAME	msgQSmLib – shared memory message queue library (VxMP Option)
ROUTINES	msgQSmCreate() – create and initialize a shared memory message queue (VxMP Option)
DESCRIPTION	<p>This library provides the interface to shared memory message queues. Shared memory message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out order. Any task running on any CPU in the system can send messages to or receive messages from a shared message queue. Tasks can also send to and receive from the same shared message queue. Full-duplex communication between two tasks generally requires two shared message queues, one for each direction.</p> <p>Shared memory message queues are created with msgQSmCreate(). Once created, they can be manipulated using the generic routines for local message queues; for more information on the use of these routines, see the manual entry for msgQLib.</p>

MEMORY REQUIREMENTS

The shared memory message queue structure is allocated from a dedicated shared memory partition. This shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared message queues, **SM_OBJ_MAX_MSG_Q**.

The message queue buffers are allocated from the shared memory system partition.

RESTRICTIONS

Shared memory message queues differ from local message queues in the following ways:

Interrupt Use:

Shared memory message queues may not be used (sent to or received from) at interrupt level.

Deletion:

There is no way to delete a shared memory message queue and free its associated shared memory. Attempts to delete a shared message queue return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Queuing Style:

The shared message queue task queueing order specified when a message queue is created must be FIFO.

CONFIGURATION

Before routines in this library can be called, the shared memory objects facility must be initialized by calling **usrSmObjInit()**. This is done automatically during VxWorks initialization if the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY

This module is distributed as a component of the unbundled shared objects memory support option, VxMP.

INCLUDE FILES

msgQSmLib.h, **msgQLib.h**, **smMemLib.h**, **smObjLib.h**

SEE ALSO

msgQLib, **smObjLib**, **msgQShow**, **usrSmObjInit()**, the VxWorks, programmer guides

mvYukonIIVxbEnd

NAME

mvYukonIIVxbEnd – Marvell Yukon II VxBus END driver

ROUTINES

ynRegister() – register with the VxBus subsystem

DESCRIPTION

This module implements a driver for the Marvell Yukon II gigabit ethernet controller devices. The Yukon II combines a 10/100/1000Mbps ethernet MAC with a Marvell 88E11xx gigabit PHY. Both copper and fiber-optic configurations are possible, though copper is more common.

The Yukon family is based partly on technology created by SysKonnect and later acquired by Marvell. The SysKonnect gigabit adapters consisted of two main components: a XaQti XMAC II serial gigabit MAC and the SysKonnect GENesis chip, which provided the PCI interface and buffer management. The GENesis was actually designed to accommodate two MACs, allowing for dual port NIC configurations. (SysKonnect originally marketed their dual port cards for failover applications only, but the hardware did allow both ports to be used as independent interfaces.)

The Yukon I family is essentially a SysKonnect GENesis combined with the Marvell GMAC into a single chip. The resulting device is programmed in much the same way as the original SysKonnect GENesis NICs, except that the MAC setup is a bit different.

With the Yukon II family, the **GENesis** portion of the controller has been modified: the DMA descriptor layouts have changed, and a status DMA descriptor ring has been added into which the controller multiplexes RX and TX completion events.

The differences in operation between the Yukon I and II devices are significant enough that it's not really desirable to combine support for both of them into a single driver. Consequently, this driver supports only the devices in the Yukon II family. This includes the Yukon II PCI-X, Yukon II PCIe, the Yukon EC PCIe and the Yukon FE devices.

The major differences between parts in the family are:

- The Yukon EC devices have only a single TX DMA queue (the other devices have two: a synchronous (normal priority) queue and an asynchronous (high priority) queue).
- The Yukon FE devices are 10/100 only and do not support jumbo frames.
- The Yukon 8022 PCI-X and 8062 PCIe devices are dual link.

The dual-link port devices contain two independent MACs and provide two sets of DMA descriptor rings and two sets of interrupt status bits. Like their SysKonnect ancestors, these devices appear as only a single PCI device. This is different from other multiport PCI network cards, which are often implemented by combining multiple standalone PCI devices together through a PCI bridge (e.g. bus0/dev0 is port 0 and bus0/dev1 is port 1), or by combining them into multiple PCI functions in a single device (e.g. bus0/dev0/func0 is port 0 and bus0/dev0/func1 is port 1).

This means that to VxBus, there will only be one device instance, however to support two devices, we need two END objects, as well as two private device contexts. To implement this, the pDrvCtrl structure attached to the VxBus device instance for a dual link device is actually an array of two pDrvCtrls. Note that this means that it's not possible to unload a single network interface on a dual link device: if the VxBus device instance is deleted, that will cause both interfaces to be removed.

Note that unlike the failover configuration supported by SysKonnect, where both ports are combined into a single virtual interface (with one port taking over for the other in the event

of a link failure), this driver allows both ports to operate as two completely independent interfaces.

The Yukon II devices support TCP/IP checksum offload and VLAN tag insertion and stripping. Currently, VLAN tag insertion and stripping is supported for RX and TX, however checksum offload is used only UDP and TCP transmit checksum acceleration.

The Yukon II gigE devices also support jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of a buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface yn0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "yn", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

Note that currently this driver has only been tested with the Marvell 88E8050 PCIe controller chip on the Intel 915GEV desktop board, and the 88E8022 on the SysKonnct SK 9S82 1000baseSX dual link adapter, however it should work with most devices in the Yukon II family.

BOARD LAYOUT The Yukon II is available on standalone PCI-X and PCIe NICs as well as integrated onto various system boards. All configurations are jumperless.

EXTERNAL INTERFACE The driver provides a vxBus external interface. The only exported routine is the **ynRegister()** function, which registers the driver with VxBus.

INCLUDE FILES **mvYukonIIVxbEnd.h mvGmac.h end.h endLib.h netBufLib.h muxLib.h**

SEE ALSO vxBus, ifLib, endLib

mvYukonVxbEnd

NAME **mvYukonVxbEnd** – Marvell Yukon I VxBus END driver

ROUTINES **ykRegister()** – register with the VxBus subsystem

DESCRIPTION This module implements a driver for the Marvell Yukon I gigabit ethernet controller devices. The Yukon I combines a 10/100/1000Mbps ethernet MAC with a Marvell 88E11xx gigabit PHY. Both copper and fiber-optic configurations are possible, though copper is more common.

The Yukon family is based partly on technology created by SysKonnnect and later acquired by Marvell. The SysKonnnect gigabit adapters consisted of two main components: a XaQti XMAC II serial gigabit MAC and the SysKonnnect GENesis chip, which provided the PCI interface and buffer management. The GENesis was actually designed to accomodate two MACs, allowing for dual port NIC configurations. (SysKonnnect originally marketed their dual port cards for failover applications only, but the hardware did allow both ports to be used as independent interfaces.)

The Yukon I family is essentially a SysKonnnect GENesis combined with the Marvell GMAC and 88E11xx PHY into a single chip. The resulting device is programmed in much the same way as the original SysKonnnect GENesis NICs, except that the MAC setup and PHY access is a bit different. Also, where the XMAC was accessed via indirect registers, the GMAC registers are mapped directly within the top level register space.

While the GENesis allowed for dual MAC configurations, only a very few dual port Yukon I devices were ever produced. Consequently, this driver is designed on only support a single port.

The Yukon gigE devices also support jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of an buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface yk0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "yk", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

BOARD LAYOUT The Yukon is available on standalone PCI and PCIe NICs as well as integrated onto various system boards. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **ykRegister()** function, which registers the driver with VxBus.

INCLUDE FILES **mvYukonVxbEnd.h mvGmac.h end.h endLib.h netBufLib.h muxLib.h**

SEE ALSO **vxBus, ifLib, endLib**

ne2000VxbEnd

NAME **ne2000VxbEnd** – NE2000 Compatible VxBus END driver

ROUTINES **eneRegister()** – register with the VxBus subsystem

DESCRIPTION This module provides driver support for NE2000 compatible 10Mbps ethernet adapters. The National Semiconductor DP83905 is used as a reference. The DP83905 is fully compliant with the IEEE 802.3u 10Base-T specification.

There are a large number of NE2000-compatible adapters in ISA, PCMCIA and PCI format, from various manufacturers, including RealTek, Winbond, ASIX electronics and Kingston Electronics. This driver uses the lowest common denominator in terms of programming API in order to support as many of them as possible. Only the basic NE2000 register set is used, along with the simple programmed I/O packet transfer scheme. No support for IFMEDIA/miiBus is included since most adapter are 10Mbps half duplex only and don't use an MII-compliant transceiver.

BOARD LAYOUT Board layout may vary depending on the manufacturer. Older ISA bus cards may have jumpers or switches for selecting the desired I/O base address and interrupt line. Newer ISA bus cards can have their I/O address and interrupt line selected via software. Typically this is done with an DOS-based command line utility. For these devices, the base address and interrupt line must be specified manually via the hwconf file in the BSP. PCI devices are jumperless and can be autoprobed, so no manual configuration is needed.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **eneRegister()** function, which registers the driver with VxBus.

For PCI devices, no special hwconf configuration should be needed, as the device will be autoprobed. ISA/local bus devices, an hwconf entry is required in order to specify the I/O base address and interrupt vector for which the device has been strapped. The hwconf resources can also be used to specify the memory size and register width of the device. An typical hwconf entry would look as follows:

```
const struct hcfResource ne2000Resources[] =
{
    { "regBase", HCF_RES_INT, {(void *)IO_ADRS_ENE} },
    { "intr", HCF_RES_INT, {(void *) (INUM_TO_IVEC(INT_NUM_ENE))} },
    { "intrLevel", HCF_RES_INT, {(void *)INT_LVL_ENE} },
    { "byteAccess", HCF_RES_INT, {(void *)0} },
    { "regWidth", HCF_RES_INT, {(void *)1} },
};
#define ne2000Num NELEMENTS(ne2000Resources)
```

HWCONF PARAMETERS

byteAccess

This controls whether the NIC RAM should be accessed using 8-bit or 16-bit operations. 16-bit accessed are used when the device has 16K of on-board RAM. This is the case on most newer devices, and consequently, *byteAccess* defaults to 0. Set it to 1 to force 8-bit operation.

regWidth

This specifies the width of each register. Normally, each register is 8 bits wide and spaced 1 byte apart, but they may be spaced 2 bytes apart depending on how the registers are mapped. The default is 1 byte. This parameter may only rarely need to be changed.

INCLUDE FILES none

SEE ALSO vxBus, ifLib, endLib, muxLib, "Writing an Enhanced Network Driver", \tb"National Semiconductor DP83905 datasheet, <http://www.national.com/ds/cgi/DP/DP83905.pdf>"

nfsCommon

NAME nfsCommon – Network File System (NFS) I/O driver

ROUTINES

- nfsMount()** – mount an NFS file system
- nfsMountAll()** – mount all file systems exported by a specified host
- nfsDevShow()** – display the mounted NFS devices
- nfsUnmount()** – unmount an NFS device
- nfsDevInfoGet()** – read configuration information from the requested device
- nfsDevListGet()** – create list of all the NFS devices in the system
- nfsDrvNumGet()** – Get driver number of NFS device
- nfsMntDump()** – display all NFS file systems mounted on a particular host
- nfsExportShow()** – display the exported file systems of a remote host
- nfsHelp()** – display the NFS help menu
- nfsAuthUnixPrompt()** – modify the NFS UNIX authentication parameters
- nfsAuthUnixShow()** – display the NFS UNIX authentication parameters
- nfsAuthUnixSet()** – set the NFS UNIX authentication parameters
- nfsAuthUnixGet()** – get the NFS UNIX authentication parameters
- nfsIdSet()** – set the ID number of the NFS UNIX authentication parameters
- nfsChkFilePerms()** – check the NFS file permissions with a given permission.
- nfsErrnoSet()** – set NFS status

DESCRIPTION This driver provides facilities for accessing files transparently over the network via NFS (Network File System). By creating a network device with **nfsMount()**, files on a remote NFS system (such as a UNIX system) can be handled as if they were local.

USER-CALLABLE ROUTINES

The **nfs2Drv()** routine initializes the NFS version 2 driver. The **nfs3Drv()** routine initializes the NFS version 3 driver. The **nfsMount()** and **nfsUnmount()** routines mount and unmount file systems. The **nfsMountAll()** routine mounts all file systems exported by a specified host.

INITIALIZATION	Before using the NFS v2 driver, it must be initialized by calling nfs2Drv() . Before using the NFS v3 driver, it must be initialized by calling nfs3Drv() . These routines must be called before any reads, writes, or other NFS calls. This is done automatically by adding the component INCLUDE_NFS2_CLIENT , INCLUDE_NFS3_CLIENT or INCLUDE_NFS_CLIENT_ALL (as appropriate) to your project.
CREATING NFS DEVICES	<p>In order to access a remote file system, an NFS device must be created by calling nfsMount(). For example, to create the device /myd0/ for the file system /d0/ on the host wrs, call:</p> <pre>nfsMount ("wrs", "/d0/", "/myd0/");</pre> <p>The file /d0/dog on the host wrs can now be accessed as /myd0/dog.</p> <p>Before mounting a file system, the host must already have been created with hostAdd(). The routine nfsDevShow() displays the mounted NFS devices.</p>
INCLUDE FILES	nfsDriver.h , ioLib.h , dirent.h
SEE ALSO	dirLib , hostAdd() , ioctl()

nfsHash

NAME	nfsHash – file based hash table for file handle to file name and reverse
ROUTINES	nfsdHashTableParamsSet() – sets up the parameters for the NFS hash table
DESCRIPTION	This library implements the hash table maintained by the NFS server.
INCLUDE FILES	none

nfsd

NAME	nfsd – NFS Server Init routines
ROUTINES	nfsdInit() – initialize the NFS server
DESCRIPTION	<p>This file implements the initialization routines for the NFS daemon.</p> <p>The nfs server is initialized by calling nfsdInit().</p>

Including any of the NFS SERVER components (**INCLUDE_NFS_SERVER_ALL**, **INCLUDE_NFS3_SERVER**, **INCLUDE_NFS2_SERVER**) enables the call to **nfsdInit()** during the boot process. So there is normally no need to call the routine manually.

The **nfsdInit()** spawns several tasks. (see details below). The **nfsd()** daemon registers the NFS services with the portmapper daemon. The registration is done depending on the components included by the user during vxWorks build.

If user includes **INCLUDE_NFS_SERVER_ALL**, nfsd will register NFS V2 and V3 services.

If user includes **INCLUDE_NFS2_SERVER** nfsd will register only NFS V2 service.

If user includes **INCLUDE_NFS3_SERVER** nfsd will register only NFS V3 service.

With the registration of NFS V3, MOUNT v1, v3 and NLM v4 are also registered.

With the registration of NFS V3 MOUNT v1 is also registered.

AUTHENTICATION AND PERMISSIONS

Currently, no authentication is done on NFS requests. **nfsdInit()** describes the authentication hooks that can be added should authentication be necessary.

Note that the DOS file system does not provide information about ownership or permissions on individual files.

TASKS

Several NFS tasks are created by **nfsdInit()**. They are:

tMountd

The mount daemon, which handles all incoming mount requests. This daemon is created by **mountdInit()**, which is called from **nfsdInit()**.

tNfsd

The NFS daemon, which queues all incoming NFS requests.

tNfsdX

The NFS request handlers, which dequeues and processes all incoming NFS requests.

INCLUDE FILES

none

nfsdCommon

NAME

nfsdCommon – Common functions for v2 and v3

ROUTINES

nfsStatusGet() – Get the statistics of the NFS server
nfsdStatusShow() – show the status of the NFS server

DESCRIPTION This file implements the common routines that will be used by the NFS version 2 and NFS version 3 procedures.

USER-CALLABLE ROUTINES
The **nfsStatusGet()** routine gets the statistis of NFS procedure calls. The **nfsdStatusShow()** routine displays the statistics of NFS procedure calls.

INCLUDE FILES none

ns8381xVxbEnd

NAME ns8381xVxbEnd – National Semiconductor DP83815/6 VxBus END driver

ROUTINES nseRegister() – register with the VxBus subsystem

DESCRIPTION This module implements a driver for the Nationam Semiconductor DP83815 MACPhyter and DP83816 MACPhyter II 10/100 PCI ethernet controllers. The DP83815 and DP83816 are fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The controller has an embedded 10/100 PHY, with MII management interface.

The MACPhyter chips use a simple list-based DMA descriptor scheme. Each decriptor is three longwords in size and contains a 32-bit buffer pointer field, a command/status field, and a next pointer field. The same descriptor format is used for both RX and TX. The device is programmed through a single 256-byte register window using either I/O space or memory mapped accesses. The chip has a single perfect filter entry for the station address and a 512-bit multicast hash table.

Note that the last two bits of RX buffer addresses are not decoded, which means that RX frame buffers must be aligned on a longword boundary. Because the ethernet frame header is only 14 bytes in size, this causes the payload to be misaligned, which can lead to unaligned accesses within the VxWorks TCP/IP stack (which uses 32-bit loads and stores to access the address fields in the IP header). On the x86, PPC and Coldfire architectures, these misaligned accesses can be safely ignored, but on all other architectures, the driver is forced to copy received buffers to fix up the alignment before passing them to the stack.

The DP83815/6 can be found on standalone NICs such as the Netgear FA311 and FA312. It's also used on some Soekris x86-based single-board computers.

BOARD LAYOUT The MACPhyter is available on standalone PCI cards as well as as integrated onto various system boards. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **nsRegister()** function, which registers the driver with VxBus.

INCLUDE FILES none

SEE ALSO vxBus, **ifLib**, "National Semiconductor DP83815 Datasheet, <http://www.national.com/ds.cgi/DP/DP83815.pdf>", "National Semiconductor DP83816 Datasheet, <http://www.national.com/ds.cgi/DP/DP83816.pdf>"

ns83902VxbEnd

NAME ns83902VxbEnd – NatSemi DP83902A ST-NIC VxBus END driver

ROUTINES **nicRegister()** – register with the VxBus subsystem

DESCRIPTION This module provides driver support for the National Semiconductor DP83902A ST-NIC 10Mbps serial network ethernet controller chip. The DP83902A is fully compliant with the IEEE 802.3u 10Base-T specification.

The DP83902A is based on the same design as the 8390, and operates in much the same fashion. The chip has 64K of internal buffer space which can be used for both RX and TX packet handling. The driver is responsible for deciding how much is reserved for each function. The buffer space is accessed via a single bi-directional I/O port, which can be accessed with either 16 bit or 8 bit operations, selectable by the driver.

There is some confusion concerning the number of available pages of memory within the chip. The manual states that each page is 256 bytes, and that the amount of memory available is "64 Kbyte (or 32 Kword)." This means that we should be able to program the BNRy register with a value of 0xFF (255 pages). However, all other known drivers use a value of 0x7F (127 pages). It's not clear why this is the case. Testing shows that using all 256 pages does in fact work with this device, so we use that here. This allows us to buffer a significant amount of RX frames, which helps avoid overruns.

The DP83902A operates in 10Mbps half duplex mode only and does not have an MII-based PHY. Consequently, this driver does not support IFMEDIA and does not provide any miiBus methods. This means it's not possible to sense link state changes.

Accessing registers on the DP83902A is a slow process: a delay must be inserted between consecutive accesses, otherwise invalid data may be written or read. The manual recommends 4 bus clocks, however the appropriate delay time may vary from one processor to the next. Also, on the SH7750 Solution Engine board, the registers are mapped at 16 bit intervals even though they are only 8 bits wide.

BOARD LAYOUT

EXTERNAL INTERFACE

INCLUDE FILES	none
SEE ALSO	vxBus, ifLib, \tb"National Semiconductor DP83902A ST-NIC datasheet, http://www.national.com/ds.cgi/DP/DP83902A.pdf "

objLib

NAME	objLib – generic object management library
ROUTINES	objShow() – show information on an object objOwnerGet() – return the object's owner objOwnerSet() – change the object's owner objClassTypeGet() – get an object's class type objNameGet() – get an object's name objNameLenGet() – get an object's name length objContextGet() – return the object's context value objContextSet() – set the object's context value objNameToId() – find object with matching name string and type
DESCRIPTION	<p>This library contains class object management routines. Classes of objects control object methods such as creation, initialization, deletion, and termination.</p> <p>Many objects in VxWorks are managed with class organization. These include tasks, semaphores, watchdogs, memory partitions, message queues, timers, and real time processes.</p> <p>To provide object management support for a system, VxWorks must be configured with the INCLUDE_OBJ_LIB component.</p> <p>If a system is configured without INCLUDE_RTP it is possible to also exclude object ownership by removing the component INCLUDE_OBJ_OWNERSHIP. This enhances the performance of object creation and deletion.</p>
INCLUDE FILE	objLib.h

objShow

NAME	objShow – wind objects show library
ROUTINES	objShowAll() – show all information on an object objHandleShow() – show information on the object referenced by an object handle objHandleTblShow() – show information on an RTP's handle table
DESCRIPTION	This library provides the routine objShowAll() to show the contents of Wind objects. The generic object information is displayed directly by objShowAll() , while the class specific information is displayed by the show routine registered for the class. The routine objShow() is invoked to display the class specific information.
CONFIGURATION	The routines in this library are included if the INCLUDE_OBJECT_SHOW component is configured into VxWorks.
INCLUDE FILES	objLib.h

partLib

NAME	partLib – routines to create disk partitions on a rawFS
ROUTINES	partLibCreate() – partition a device xbdCreatePartition() – partition an XBD device
DESCRIPTION	This library contains routines for handling partitions.
EXAMPLE	The following code will initialize a disk which is expected to have up to 4 partitions. The parameter passed in is the base name of the device to be partitioned. E.g. "/ata00". Partitions are automatically named using the base name followed by a ":" and then the partition number. Using the fsMonitor component, these automatic names can be mapped to something more meaningful:

```

STATUS usrCreatePartitions( char *devName )
{
    devname_t  baseName;
    char       autoPartName[16];
    char *     newPartName[4] = {"/p1", "/p2", "/p3", "/p4"};
    STATUS     result;
    int        i, fd;

    /* Name mapping */

```

```

    /* Get the base name of the device */
    fd = open (devName, 0, 0666);
    if (fd < 0)
    {
        return (ERROR);
    }

    ioctl (fd, XBD_GETBASENAME, (int)baseName);

    close (fd);

    for (i = 0; i < 4; i++)
    {
        sprintf (autoPartName, "%s:%d", baseName, i+1);
        printf ("Installing mapping from %s to %s\n", autoPartName,
            newPartName[i]);
        fsmNameInstall (autoPartName, newPartName[i]);
    }

    /* create 4 partitions on the device all with equal sizes */

    result = xbdCreatePartition ( devName, 4, 25, 25, 25 );

    if (result != OK)
    {
        return (ERROR);
    }

    /* create file systems atop each partition */

    dosFsVolFormat ( newPartName[0], 0, NULL);
    dosFsVolFormat ( newPartName[1], 0, NULL);
    dosFsVolFormat ( newPartName[2], 0, NULL);
    dosFsVolFormat ( newPartName[3], 0, NULL);

    return (OK);
}

```

INCLUDE FILES **fsMonitor.h, fsEventUtilLib.h, drv/xbd/xbd.h, drv/erf/erfLib.h**

SEE ALSO **dosFsLib, fsMonitor**

passFsLib

NAME **passFsLib** – pass-through file system library (VxSim)

ROUTINES **passFsDevInit()** – associate a device with passFs file system functions
passFsInit() – prepare to use the passFs library

DESCRIPTION This library implements a file-oriented device driver for VxSim to provide an easy access to the host file system. This device driver is named pass-through file system (passFs). In general, the routines are not to be called directly by users, but rather by the VxWorks I/O System. All the host hard drives can be accessed by only one passFs device.

USING THIS LIBRARY

The various routines provided by passFs may be separated into two groups: device initialization and file system operation.

The **passFsInit()** and **passFsDevInit()** APIs are the principal initialization functions. They should be called once during system initialization. The initialization is done automatically when **INCLUDE_PASSFS** component is defined. The **PASSFS_CACHE** parameter of **INCLUDE_PASSFS** component allow to enable or disable passFs cache, by default it is enabled.

I/O is performed on this device driver exactly as it would be on any device referencing a VxWorks file system. File operations, such as **read()** and **write()**, are then executed on the host file system.

INITIALIZING PASSFSLIB

Before using any other routines in **passFsLib**, the routine **passFsInit()** must be called to initialize this library. First argument specifies the number of passFs devices that may be open at once, second argument is a boolean that specifies if cache must be enabled or not. The **passFsDevInit()** routine associates a device name with the **passFsLib** functions. The parameter expected by **passFsDevInit()** is a pointer to a string which specifies the device name. This device name will be part of the pathname for I/O operations which operates on the device. It will appear in the I/O system device table, which may be displayed using the **iosDevShow()** routine.

As an example:

```
passFsInit (1, 1);  
passFsDevInit ("host:");
```

After the **passFsDevInit()** call, when **passFsLib** receives a request from the I/O system, it calls the host Operating system I/O system to service the request. Only one device can be created.

The default passFs device name is "host:" for Windows VxSim and "host name:" on UNIX VxSim. It can be changed by using the VxSim command line option:

```
vxsim -hn <host name> | -hostname <host name>
```

Then, *host name* parameter will be used as passFs device name.

PATH SEPARATOR VxSim passFs is using Unix like path separator, even on Windows VxSim.

PATH NAME On Windows VxSim, the VxWorks syntax to access a host file system is :

```
<passFs device name>:<disk>:/dir1/dir2/file.      or  
/<disk>/dir1/dir2/file.
```

For example, to open the host file "c:\myDir\mySubDir\myFile" :

```
open ("host:c:/myDir/mySubDir/myFile", O_RDWR, 0); or  
open ("/c/myDir/mySubDir/myFile", O_RDWR, 0);
```

On UNIX VxSim, the VxWorks syntax to access a host file system is :

```
<passFs device name>:/dir1/dir2/file.
```

For example, if VxSim is running on "mySolarisStation" host, to open the host file
"/myHome/mySubDir/myFile" :

```
open ("mySolarisStation:/myHome/mySubDir/myFile", O_RDWR, 0);
```

READING DIRECTORY ENTRIES

All directory functions, such as **mkdir()**, **rmdir()**, **opendir()**, **readdir()**, **losedir()**, and **rewinddir()** are supported by passFs.

FILE INFORMATION

To obtain more detailed information about a specific file, use the **fstat()** or **stat()** function. Along with standard file information, the structure used by these routines also returns the file attribute byte from a passFs directory entry.

FILE SYSTEM INFORMATION

To obtain more detailed information about a specific file system, use the **fstatfs()** or **statfs()** functions.

FILE DATE AND TIME

Host OS file date and time are passed though to VxWorks.

FLAGS

Standard I/O flags (**O_RDWR**, **O_RDONLY**, ..) convention is handled by passFs. VxWorkd I/O flags are converted to passFs flags and then to host OS specific flags.

RESTRICTION

rename() and **fstatfs()** APIs are not supported on Windows VxSim because the Windows **diskFormat()** API is not supported.
limitations.

INCLUDE FILES

passFsLib.h

SEE ALSO

ioLib, **iosLib**, **dirLib**, **ramDrv**

pentiumALib

NAME

pentiumALib – P5, P6 and P7 family processor specific routines

ROUTINES

pentiumCr4Get() – get contents of CR4 register
pentiumCr4Set() – sets specified value to the CR4 register
pentiumP6PmcStart() – start both PMC0 and PMC1
pentiumP6PmcStop() – stop both PMC0 and PMC1
pentiumP6PmcStop1() – stop PMC1
pentiumP6PmcGet() – get the contents of PMC0 and PMC1
pentiumP6PmcGet0() – get the contents of PMC0
pentiumP6PmcGet1() – get the contents of PMC1
pentiumP6PmcReset() – reset both PMC0 and PMC1
pentiumP6PmcReset0() – reset PMC0
pentiumP6PmcReset1() – reset PMC1
pentiumP5PmcStart0() – start PMC0
pentiumP5PmcStart1() – start PMC1
pentiumP5PmcStop() – stop both P5 PMC0 and PMC1
pentiumP5PmcStop0() – stop P5 PMC0
pentiumP5PmcStop1() – stop P5 PMC1
pentiumP5PmcGet() – get the contents of P5 PMC0 and PMC1
pentiumP5PmcGet0() – get the contents of P5 PMC0
pentiumP5PmcGet1() – get the contents of P5 PMC1
pentiumP5PmcReset() – reset both PMC0 and PMC1
pentiumP5PmcReset0() – reset PMC0
pentiumP5PmcReset1() – reset PMC1
pentiumTscGet64() – get 64Bit TSC (Timestamp Counter)
pentiumTscGet32() – get the lower half of the 64Bit TSC (Timestamp Counter)
pentiumTscReset() – reset the TSC (Timestamp Counter)
pentiumMsrGet() – get the contents of the specified MSR (Model Specific Register)
pentiumMsrSet() – set a value to the specified MSR (Model Specific Registers)
pentiumTlbFlush() – flush TLBs (Translation Lookaside Buffers)
pentiumSerialize() – execute a serializing instruction CPUID
pentiumBts() – execute atomic compare-and-exchange instruction to set a bit
pentiumBtc() – execute atomic compare-and-exchange instruction to clear a bit

DESCRIPTION This module contains Pentium and PentiumPro specific routines written in assembly language.

MCA (Machine Check Architecture)

The Pentium processor introduced a new exception called the machine-check exception (interrupt-18). This exception is used to signal hardware-related errors, such as a parity error on a read cycle. The PentiumPro processor extends the types of errors that can be detected and that generate a machine-check exception. It also provides a new machine-check architecture that records information about a machine-check error and provides the basis for an extended error logging capability.

MCA is enabled and its status registers are cleared zero in **sysHwInit()**. Its registers are accessed by **pentiumMsrSet()** and **pentiumMsrGet()**.

PMC (Performance Monitoring Counters)

The P5 and P6 family of processor has two performance-monitoring counters for use in monitoring internal hardware operations. These counters are duration or event counters that can be programmed to count any of approximately 100 different types of events, such as the number of instructions decoded, number of interrupts received, or number of cache loads. However, the set of events can be counted with PMC is different in the P5 and P6 family of processors; and the locations and bit definitions of the related counter and control registers are also different. So there are two set of PMC routines, one for P6 family and one for p5 family respectively.

There are nine routines to interface the PMC of P6 family processors. These nine routines are:

```
STATUS pentiumP6PmcStart
(
    int pmcEvtSel0;          /* performance event select register 0 */
    int pmcEvtSel1;          /* performance event select register 1 */
)

void pentiumP6PmcStop (void)
void pentiumP6PmcStop1 (void)
void pentiumP6PmcGet
(
    long long int * pPmc0; /* performance monitoring counter 0 */
    long long int * pPmc1; /* performance monitoring counter 1 */
)

void pentiumP6PmcGet0
(
    long long int * pPmc0; /* performance monitoring counter 0 */
)

void pentiumP6PmcGet1
(
    long long int * pPmc1; /* performance monitoring counter 1 */
)

void pentiumP6PmcReset (void)
void pentiumP6PmcReset0 (void)
void pentiumP6PmcReset1 (void)
```

pentiumP6PmcStart() starts both PMC0 and PMC1. **pentiumP6PmcStop()** stops them, and **pentiumP6PmcStop1()** stops only PMC1. **pentiumP6PmcGet()** gets contents of PMC0 and PMC1. **pentiumP6PmcGet0()** gets contents of PMC0, and **pentiumP6PmcGet1()** gets contents of PMC1. **pentiumP6PmcReset()** resets both PMC0 and PMC1. **pentiumP6PmcReset0()** resets PMC0, and **pentiumP6PmcReset1()** resets PMC1. PMC is enabled in **sysHwInit()**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

There are ten routines to interface the PMC of P5 family processors. These ten routines are:

```
STATUS pentiumP5PmcStart0
(
    int pmc0Cesr; /* PMC0 control and event select */
)

STATUS pentiumP5PmcStart1
```

```

    (
        int pmc1Cesr; /* PMC1 control and event select */
    )
void    pentiumP5PmcStop0 (void)
void    pentiumP5PmcStop1 (void)
void    pentiumP5PmcGet
    (
        long long int * pPmc0; /* performance monitoring counter 0 */
        long long int * pPmc1; /* performance monitoring counter 1 */
    )
void    pentiumP5PmcGet0
    (
        long long int * pPmc0; /* performance monitoring counter 0 */
    )
void    pentiumP5PmcGet1
    (
        long long int * pPmc1; /* performance monitoring counter 1 */
    )
void    pentiumP5PmcReset (void)
void    pentiumP5PmcReset0 (void)
void    pentiumP5PmcReset1 (void)

```

pentiumP5PmcStart0() starts PMC0, and **pentiumP5PmcStart1()** starts PMC1.

pentiumP5PmcStop0() stops PMC0, and **pentiumP5PmcStop1()** stops PMC1.

pentiumP5PmcGet() gets contents of PMC0 and PMC1. **pentiumP5PmcGet0()** gets contents of PMC0, and **pentiumP5PmcGet1()** gets contents of PMC1.

pentiumP5PmcReset() resets both PMC0 and PMC1. **pentiumP5PmcReset0()** resets PMC0, and **pentiumP5PmcReset1()** resets PMC1. PMC is enabled in **sysHwInit()**.

Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

MSR (Model Specific Register)

The concept of model-specific registers (MSRs) to control hardware functions in the processor or to monitor processor activity was introduced in the PentiumPro processor. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine check architecture, and the MTRRs. The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

There are two routines to interface the MSR. These two routines are:

```

void    pentiumMsrGet
    (
        int address,          /* MSR address */
        long long int * pData /* MSR data */
    )

void    pentiumMsrSet
    (
        int address,          /* MSR address */
        long long int * pData /* MSR data */
    )

```

pentiumMsrGet() get contents of the specified MSR, and **pentiumMsrSet()** sets value to the specified MSR.

TSC (Time Stamp Counter)

The PentiumPro processor provides a 64-bit time-stamp counter that is incremented every processor clock cycle. The counter is incremented even when the processor is halted by the HLT instruction or the external STPCLK# pin. The time-stamp counter is set to 0 following a hardware reset of the processor. The RDTSC instruction reads the time stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for 64-bit counter wraparound. Intel guarantees, architecturally, that the time-stamp counter frequency and configuration will be such that it will not wraparound within 10 years after being reset to 0. The period for counter wrap is several thousands of years in the PentiumPro and Pentium processors.

There are three routines to interface the TSC. These three routines are:

```
void pentiumTscReset (void)

void pentiumTscGet32 (void)

void pentiumTscGet64
(
    long long int * pTsc    /* TSC */
)
```

pentiumTscReset() resets the TSC. **pentiumTscGet32()** gets the lower half of the 64Bit TSC, and **pentiumTscGet64()** gets the entire 64Bit TSC.

Four other routines are provided in this library. They are:

```
void pentiumTlbFlush (void)

void pentiumSerialize (void)

STATUS pentiumBts
(
    char * pFlag                /* flag address */
)

STATUS pentiumBtc (pFlag)
(
    char * pFlag                /* flag address */
)
```

pentiumTlbFlush() flushes TLBs (Translation Lookaside Buffers). **pentiumSerialize()** does serialization by executing CPUID instruction. **pentiumBts()** executes an atomic compare-and-exchange instruction to set a bit. **pentiumBtc()** executes an atomic compare-and-exchange instruction to clear a bit.

INCLUDE FILES none

SEE ALSO *Pentium, PentiumPro Family Developer's Manual*

pentiumLib

NAME	pentiumLib – Pentium and Pentium[234] library
ROUTINES	pentiumMtrrEnable() – enable MTRR (Memory Type Range Register) pentiumMtrrDisable() – disable MTRR (Memory Type Range Register) pentiumMtrrGet() – get MTRRs to a specified MTRR table pentiumMtrrSet() – set MTRRs from specified MTRR table with WRMSR instruction. pentiumPmcStart() – start both PMC0 and PMC1 pentiumPmcStart0() – start PMC0 pentiumPmcStart1() – start PMC1 pentiumPmcStop() – stop both PMC0 and PMC1 pentiumPmcStop0() – stop PMC0 pentiumPmcStop1() – stop PMC1 pentiumPmcGet() – get the contents of PMC0 and PMC1 pentiumPmcGet0() – get the contents of PMC0 pentiumPmcGet1() – get the contents of PMC1 pentiumPmcReset() – reset both PMC0 and PMC1 pentiumPmcReset0() – reset PMC0 pentiumPmcReset1() – reset PMC1 pentiumMsrInit() – initialize all the MSRs (Model Specific Register) pentiumMcaEnable() – enable/disable the MCA (Machine Check Architecture)
DESCRIPTION	This library provides Pentium and Pentium[234] specific routines.

MTRR (Memory Type Range Register)

MTRR (Memory Type Range Register) are a new feature introduced in the P6 family processor that allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped IO. MTRRs configure an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), write-combining (WC), write-through (WT), write-protected (WP), and write-back (WB).

There is one table - `sysMtrr[]` in **sysLib.c** - and four routines to interface the MTRR. These four routines are:

```
void pentiumMtrrEnable (void)

void pentiumMtrrDisable (void)
```

```
STATUS pentiumMtrrGet
(
    MTRR * pMtrr          /* MTRR table */
)

STATUS pentiumMtrrSet (void)
(
    MTRR * pMtrr          /* MTRR table */
)
```

pentiumMtrrEnable() enables MTRR, **pentiumMtrrDisable()** disables MTRR. **pentiumMtrrGet()** gets MTRRs to the specified MTRR table. **pentiumMtrrSet()** sets MTRRs from the specified MTRR table. The MTRR table is defined as follows:

```
typedef struct mtrr_fix          /* MTRR - fixed range register */
{
    char type[8];                /* address range: [0]=0-7 ... [7]=56-63 */
} MTRR_FIX;

typedef struct mtrr_var          /* MTRR - variable range register */
{
    long long int base;          /* base register */
    long long int mask;          /* mask register */
} MTRR_VAR;

typedef struct mtrr              /* MTRR */
{
    int cap[2];                  /* MTRR cap register */
    int deftype[2];              /* MTRR defType register */
    MTRR_FIX fix[11];            /* MTRR fixed range registers */
    MTRR_VAR var[8];             /* MTRR variable range registers */
} MTRR;
```

Fixed Range Register's type array can be one of following memory types. **MTRR_UC** (uncacheable), **MTRR_WC** (write-combining), **MTRR_WT** (write-through), **MTRR_WP** (write-protected), and **MTRR_WB** (write-back). MTRR is enabled in **sysHwInit()**.

PMC (Performance Monitoring Counters)

The P5 and P6 family of processors has two performance-monitoring counters for use in monitoring internal hardware operations. These counters are duration or event counters that can be programmed to count any of approximately 100 different types of events, such as the number of instructions decoded, number of interrupts received, or number of cache loads. However, the set of events can be counted with PMC is different in the P5 and P6 family of processors; and the locations and bit definitions of the related counter and control registers are also different. So there are two set of PMC routines, one for P6 family and one for P5 family respectively in **pentiumALib**. For convenience, the PMC routines here are acting as wrappers to those routines in **pentiumALib**. They will call the P5 or P6 routine depending on the processor type.

There are twelve routines to interface the PMC. These twelve routines are:

```
STATUS pentiumPmcStart
(
```

```

        int pmcEvtSel0;          /* performance event select register 0 */
        int pmcEvtSel1;          /* performance event select register 1 */
    )
    STATUS pentiumPmcStart0
    (
        int pmcEvtSel0;          /* performance event select register 0 */
    )
    STATUS pentiumPmcStart1
    (
        int pmcEvtSel1;          /* performance event select register 1 */
    )
    void pentiumPmcStop (void)
    void pentiumPmcStop0 (void)
    void pentiumPmcStop1 (void)
    void pentiumPmcGet
    (
        long long int * pPmc0; /* performance monitoring counter 0 */
        long long int * pPmc1; /* performance monitoring counter 1 */
    )
    void pentiumPmcGet0
    (
        long long int * pPmc0; /* performance monitoring counter 0 */
    )
    void pentiumPmcGet1
    (
        long long int * pPmc1; /* performance monitoring counter 1 */
    )
    void pentiumPmcReset (void)
    void pentiumPmcReset0 (void)
    void pentiumPmcReset1 (void)

```

pentiumPmcStart() starts both PMC0 and PMC1. **pentiumPmcStart0()** starts PMC0, and **pentiumPmcStart1()** starts PMC1. **pentiumPmcStop()** stops both PMC0 and PMC1.

pentiumPmcStop0() stops PMC0, and **pentiumPmcStop1()** stops PMC1.

pentiumPmcGet() gets contents of PMC0 and PMC1. **pentiumPmcGet0()** gets contents of PMC0, and **pentiumPmcGet1()** gets contents of PMC1. **pentiumPmcReset()** resets both PMC0 and PMC1. **pentiumPmcReset0()** resets PMC0, and **pentiumPmcReset1()** resets PMC1. PMC is enabled in **sysHwInit()**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

MSR (Model Specific Registers)

The P5(Pentium), P6(PentiumPro, II, III), and P7(Pentium4) family processors contain a model-specific registers (MSRs). These registers are implementation specific. They are provided to control a variety of hardware and software related features including the performance monitoring, the debug extensions, the machine check architecture, etc.

There is one routine - **pentiumMsrInit()** - to initialize all the MSRs. This routine initializes all the MSRs in the processor and works on either P5, P6 or P7 family processors.

MCA (Machine Check Architecture)

The P5(Pentium), P6(PentiumPro, II, III), and P7(Pentium4) family processors have a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as system bus errors, ECC errors, parity errors, cache errors and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs for recording errors that are detected. The processor signals the detection of a machine-check error by generating a machine-check exception, which an abort class exception. The implementation of the machine-check architecture, does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check exception handler can collect information about the machine-check error from the machine-check MSRs.

There is one routine - **pentiumMcaEnable()** - to enable or disable the MCA. The routine enables or disables 1) the Machine Check Architecture and its Error Reporting register banks 2) the Machine Check Exception by toggling the MCE bit in the CR4. This routine works on either P5, P6 or P7 family.

INCLUDE FILES none

SEE ALSO *PentiumALib, Pentium, Pentium[234] Family Developer's Manual*

pentiumShow

NAME **pentiumShow** – Pentium and Pentium[234] specific show routines

ROUTINES **pentiumMcaShow()** – show MCA (Machine Check Architecture) registers
pentiumPmcShow() – show PMCs (Performance Monitoring Counters)
pentiumMsrShow() – show all the MSR (Model Specific Register)

DESCRIPTION This library provides Pentium and Pentium[234] specific show routines.

pentiumMcaShow() shows Machine Check Global Control Registers and Error Reporting Register Banks. **pentiumPmcShow()** shows PMC0 and PMC1, and reset them if the parameter zap is **TRUE**.

INCLUDE FILES none

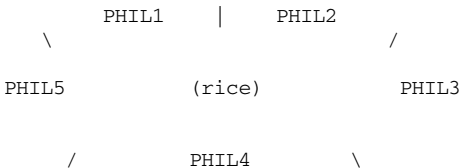
SEE ALSO *VxWorks Programmer's Guide: Configuration*

phil

NAME	phil – VxWorks/SMP Dijkstra's dining philosophers demo
ROUTINES	philDemo() – entry point for VxWorks/SMP Dijkstra's dining philosophers demo
DESCRIPTION	This program demonstrates Dijkstra's dining philosophers problem (see "Cooperating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, (1965)). It is considered a classic process synchronization problem. This demo uses sempahores for synchronization.

Dining Philosophers Problem Description

Five philosophers spend their lives thinking and eating. They share a common table. Each philosopher has their own chair. At the center of the table is a bowl of rice. The table is laid with five chopsticks (see figure below). When a philosopher thinks, s/he does not eat, and vice versa. When a philosopher is hungry, s/he tries to pick up the two chopsticks that are closest. S/he may only pick up one stick at a time. When s/he has both chopsticks, s/he eats without releasing his chopsticks. When s/he is finished eating, s/he puts down both chopsticks and starts thinking.



The original version of **phil.c** was written by James R. Pickering. It has been ported by Wind River Systems to VxWorks. The version ported to VxWorks uses binary sempahores, and thus serves as programming example for VxWorks binary sempahores.

This demo also illustrates the portability of applications from the uniprocessor version of VxWorks to the SMP version of VxWorks. This demo was created from the existing (uniprocessor) VxWorks 6.x RTP sample application shipped in the directory `$WIND_BASE/target/usr/apps/samples/philosophers`. The only modification performed to create the SMP version of **phil.c** occured in the entry point code; RTPs require a "main (int argc, char * argc[])" entry point, whereas kernel applications do not.

ANSI escape sequences are used to print bold/underline text and to perform cursor movements. Thus the console terminal program, e.g. HyperTerminal for PCs, must support ANSI escape sequences, and should be sized with 25 rows (or more), and 80 columns (or more).

To run the demo, execute the entry point function **philDemo()** from the target shell. Specifying a non-0 value for *arg* disables the usage of ANSI escape sequences in the console output. The demo runs indefinitely.

The following console output is an example of executing the demo specifying that ANSI escape sequences are to be used. The following output was obtained from a Freescale hpcNet8641 (8641D dual-core processor) board:

```
->philDemo 0 (or no parameter)

Phil - 1 is eating... on CPU 1
                                     Phil - 2 is hungry on CPU 1

Phil - 5 is thinking... on CPU 1
                                     Phil - 3 is thinking... on CPU 0

                                     Phil - 4 is eating... on CPU 0
```

The following console output is an example of executing the demo specifying that ANSI escape sequences are not to be used. Again, the following output was obtained from a Freescale hpcNet8641 (8641D dual-core processor) board:

```
-> philDemo 1
Phil - 1 is thinking... on CPU 0
Phil - 1 is hungry on CPU 0
Phil - 1 is hungry (right) on CPU 0
Phil - 1 is eating... on CPU 0
Phil - 2 is thinking... on CPU 0
Phil - 2 is hungry on CPU 0
Phil - 3 is thinking... on CPU 0
Phil - 3 is hungry on CPU 0
Phil - 3 is hungry (right) on CPU 0
Phil - 3 is eating... on CPU 0
Phil - 4 is thinking... on CPU 0
Phil - 4 is hungry on CPU 0
Phil - 5 is thinking... on CPU 0
Phil - 5 is hungry on CPU 0
Phil - 1 is thinking... on CPU 1
Phil - 2 is hungry (left) on CPU 0
Phil - 5 is hungry (right) on CPU 0
Phil - 5 is eating... on CPU 1
Phil - 4 is hungry (left) on CPU 0
Phil - 2 is eating... on CPU 1
Phil - 3 is thinking... on CPU 0
Phil - 1 is hungry on CPU 1
Phil - 2 is thinking... on CPU 0
Phil - 1 is hungry (right) on CPU 1
Phil - 4 is eating... on CPU 0
Phil - 1 is eating... on CPU 1
```

```
Phil - 5 is thinking... on CPU 0
Phil - 3 is hungry on CPU 0
Phil - 1 is thinking... on CPU 1
Phil - 1 is hungry on CPU 1
Phil - 1 is hungry (right) on CPU 1
Phil - 1 is eating... on CPU 1
Phil - 5 is hungry on CPU 0
Phil - 1 is thinking... on CPU 1
Phil - 5 is hungry (right) on CPU 0
Phil - 1 is hungry on CPU 1
Phil - 1 is hungry (right) on CPU 1
Phil - 3 is hungry (right) on CPU 1
Phil - 4 is thinking... on CPU 0
Phil - 3 is eating... on CPU 1
Phil - 5 is eating... on CPU 0
Phil - 2 is hungry on CPU 1
Phil - 1 is eating... on CPU 1
Phil - 5 is thinking... on CPU 0
Phil - 3 is thinking... on CPU 1
```

INCLUDE FILES none

pipeDrv

NAME pipeDrv – pipe I/O driver

ROUTINES **pipeDrv()** – initialize the pipe driver
 pipeDevCreate() – create a pipe device
 pipeDevDelete() – delete a pipe device

DESCRIPTION The pipe driver provides a mechanism that lets tasks communicate with each other through the standard I/O interface. Pipes can be read and written with normal **read()** and **write()** calls. The pipe driver is initialized with **pipeDrv()**. Pipe devices are created with **pipeDevCreate()**.

The pipe driver uses the VxWorks message queue facility to do the actual buffering and delivering of messages. The pipe driver simply provides access to the message queue facility through the I/O system. The main differences between using pipes and using message queues directly are:

- pipes are named (with I/O device names).
- pipes use the standard I/O functions -- **open()**, **close()**, **read()**, **write()** -- while message queues use the functions **msgQSend()** and **msgQReceive()**.
- pipes respond to standard **ioctl()** functions.
- pipes can be used in a **select()** call.

- message queues have more flexible options for timeouts and message priorities.
- pipes are less efficient than message queues because of the additional overhead of the I/O system.

CONFIGURATION To use the pipe driver library, configure VxWorks with the **INCLUDE_PIPES** component.

INSTALLING THE DRIVER

Before using the driver, it must be initialized and installed by calling **pipeDrv()**. This routine must be called before any pipes are created. It is called automatically when VxWorks is configured with the **INCLUDE_PIPES** component.

CREATING PIPES Before a pipe can be used, it must be created with **pipeDevCreate()**. For example, to create a device pipe `"/pipe/demo"` with up to 10 messages of size 100 bytes, the proper call is:

```
pipeDevCreate ("/pipe/demo", 10, 100);
```

USING PIPES Once a pipe has been created it can be opened, closed, read, and written just like any other I/O device. Often the data that is read and written to a pipe is a structure of some type. Thus, the following example writes to a pipe and reads back the same data:

```
{
int fd;
struct msg outMsg;
struct msg inMsg;
int len;

fd = open ("/pipe/demo", O_RDWR);

write (fd, &outMsg, sizeof (struct msg));
len = read (fd, &inMsg, sizeof (struct msg));

close (fd);
}
```

The data written to a pipe is kept as a single message and will be read all at once in a single read. If **read()** is called with a buffer that is smaller than the message being read, the remainder of the message will be discarded. Thus, pipe I/O is "message oriented" rather than "stream oriented." In this respect, VxWorks pipes differ significantly from UNIX pipes which are stream oriented and do not preserve message boundaries.

Open pipe file descriptors do not honor the flags or mode values. Any open pipe can always be read or written regardless of the flag value used to open the file (**O_RDONLY** or **O_WRONLY**).

WRITING TO PIPES FROM INTERRUPT SERVICE ROUTINES

Interrupt service routines (ISR) can write to pipes, providing one of several ways in which ISRs can communicate with tasks. For example, an interrupt service routine may handle the time-critical interrupt response and then send a message on a pipe to a task that will continue with the less critical aspects. However, the use of pipes to communicate from an

ISR to a task is now discouraged in favor of the direct message queue facility, which offers lower overhead (see the reference entry for **msgQLib** for more information).

SELECT CALLS An important feature of pipes is their ability to be used in a **select()** call. The **select()** routine allows a task to wait for input from any of a selected set of I/O devices. A task can use **select()** to wait for input from any combination of pipes, sockets, or serial devices. See the reference entry for **select()**.

IOCTL FUNCTIONS

Pipe devices respond to the following **ioctl()** functions. These functions are defined in the header file **ioLib.h**.

FIOGETNAME

Gets the file name of *fd* and copies it to the buffer referenced by *nameBuf*:

```
status = ioctl (fd, FIOGETNAME, &nameBuf);
```

FIONREAD

Copies to *nBytesUnread* the number of bytes remaining in the first message in the pipe:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

FIONMSGS

Copies to *nMessages* the number of discrete messages remaining in the pipe:

```
status = ioctl (fd, FIONMSGS, &nMessages);
```

FIOFLUSH

Discards all messages in the pipe and releases the memory block that contained them:

```
status = ioctl (fd, FIOFLUSH, 0);
```

INCLUDE FILES **ioLib.h**, **pipeDrv.h**

SEE ALSO **select()**, **msgQLib**, the VxWorks programmer guides.

pmLib

NAME **pmLib** – persistent memory library

ROUTINES

- pmFreeSpace()** – returns the amount of free space left in the PM arena
- pmRegionOpen()** – opens an existing persistent heap region
- pmRegionCreate()** – creates a persistent heap region
- pmRegionClose()** – closes a region making it inaccessible to clients
- pmRegionProtect()** – makes a PM region read-only
- pmRegionAddr()** – returns the address of a persistent heap region
- pmRegionSize()** – return the size of a persistent heap region

pmLib

pmValidate() – validates a PM arena

pmInvalidate() – invalidates the entire PM arena

pmShow() – shows the created persistent heap segments

DESCRIPTION

This library provides an interface to a region of memory which is intended to survive across reboots. This region is known as the **persistent memory arena**. This arena can be used to allocate individual regions within it that are owned by different parts of the OS (e.g. WindView post-mortem logs, ED&R error-logs, etc).

The arena header is verified for consistency using a magic number and a checksum, and is protected against over-writes by the MMU. It is also possible to write-protect each region, again by using the generic MMU/**vmBaseLib** API.

On creation, an arena can be configured to be writeable (i.e. with mode **PM_PROT_RDWR**), although by default an arena would be write-protected (i.e. with mode **PM_PROT_RDONLY**).

A system can have any number of arenas, although typically only one is configured. All the public **pmLib** functions take a **PM_arena_def** as their first argument, and this typedef is actually a function pointer, to a function which returns the start address of the arena itself, and also returns (by reference) the arena size. The reason for this is that functions are guaranteed to be immutable, and so **pmLib** contains no global variables at all. This is a necessity for maintaining protection of the arena under all circumstances, since global variables always face the possibility of being inadvertently over-written.

A default arena is defined by the function **pmDefaultArena** which can be over-ridden in **usrPmInit.c** if necessary, for example, to place the default PM arena in non-volatile memory for a specific BSP.

Users of **pmLib** can also provide their own arena definition functions, as long as they conform to the prototype **PM_arena_def** in **pmLib.h**.

CONFIGURATION

To use the persistent memory library, configure VxWorks with the **INCLUDE_EDR_PM** component.

TYPICAL USAGE MODE

The typical way of using **pmLib** is that the clients who wish to allocate persistent regions do so at boot time. Typically, they would try to open an existing region (by name), and if that fails they would create a new region.

Once a region has been created there is no way to **destroy** it, since **pmLib** does not provide a heap, as such. Rather, the only way to start over is to call **pmInvalidate()**, which renders the entire PM arena (all of **USER_RESERVED_MEM** in a typical configuration) invalid. Thus, on the next reboot, the arena will be re-initialized and all clients will be able to re-create their regions afresh.

INCLUDE FILES

none

poolLib

NAME	poolLib – Memory Pool Library
ROUTINES	poolCreate() – create a pool poolDelete() – delete a pool poolBlockAdd() – add an item block to the pool poolUnusedBlocksFree() – free blocks that have all items unused poolItemGet() – get next free item from pool and return a pointer to it poolItemReturn() – return an item to the pool poolIncrementSet() – set the increment value used to grow the pool poolIncrementGet() – get the increment value used to grow the pool poolTotalCount() – return total number of items in pool poolFreeCount() – return number of free items in pool
DESCRIPTION	<p>This module contains the Memory Pool library. Pools provide a fast and efficient memory management when an application uses a large number of identically sized memory items (e.g. structures, objects) by minimizing the number of allocations from a memory partition. The use of pools also reduces possible fragmentation caused by frequent memory allocation and freeing.</p> <p>A pool is a dynamic set of statically sized memory items. All items in a pool are of the same size, and all are guaranteed a power of two alignment. The size and alignment of items are specified at pool creation time. An item can be of arbitrary size, but the actual memory used up by each item is at least 8 bytes, and it is a multiple of the item alignment. The minimum alignment of items is the architecture specific allocation alignment.</p> <p>Pools are created and expanded using a specified number of items for initial size and another number of items for incremental pool additions. The initial set of items and the incremental pool items are added as one block of memory. Each memory block can be allocated from either the system memory partition (when the partition ID passed to poolCreate() is NULL), a user-provided memory partition. A block can be also added to the pool using any memory specified by the user using poolBlockAdd(). For example, if all items in a pool have to be in some specific memory zone, the pool can be created with initial and incremental item count as zero in order to prevent automatic creation of blocks from memory partitions, and explicitly adding blocks with poolBlockAdd() as needed. The memory provided to the pool must be writable. Allocation and free from memory pools are performed using the poolItemGet() and poolItemReturn() routines.</p> <p>If the pool item increment is specified as zero, the pool will be static, unable to grow dynamically. A static pool is more deterministic.</p> <p>Pools are intended for use in systems requiring frequent allocating and freeing of memory in statically sized blocks such as used in messaging systems, data-bases, and the like. This pool system is dynamic and grows upon request, eventually allowing a system to achieve a stable state with no further memory requests needed.</p>

The **poolItemGet()** and **poolItemReturn()** functions may be called from interrupt context as long as the pool was created without the **POOL_THREAD_SAFE**. Other routines provided by this library should not be called from an ISR.

Pool system statistics are available for specific pools as well as the overall pool system. These show routines are available only if **INCLUDE_SHOW_ROUTINES** is defined.

INCLUDE FILE	poolLib.h
SEE ALSO	memPartLib , poolShow

poolShow

NAME	poolShow – Wind Memory Pool Show Library
ROUTINES	poolShow() – display pool information
DESCRIPTION	<p>This library provides routines which display information about memory pools in the system, and detailed statistics about individual memeory pools.</p> <p>To include this library, select the INCLUDE_POOL_SHOW component.</p>
INCLUDE FILES	none
SEE ALSO	poolLib

primesDemo

NAME	primesDemo – VxWorks SMP prime number computation demo
ROUTINES	primesCompute() – entry point for the VxWorks SMP prime number computation demo
DESCRIPTION	<p>This module implements a VxWorks SMP prime number computation demo. This demo is an example of a number crunching application, as opposed to an I/O intensive and/or kernel system call intensive application. In addition, the various computational tasks (and thus CPUs) cooperate to achieve a common goal.</p> <p>The prime numbers are computed using the "Sieve of Eratosthenes" algorithm. The following is a description of the algorithm from Wikipedia:</p>

"In mathematics, the Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer. It is the predecessor to the modern Sieve of Atkin, which is faster but more complex. It was created by Eratosthenes, an ancient Greek mathematician. Wheel factorization is often applied on the list of integers to be checked for primality, before Sieve of Eratosthenes is used, to increase the speed.

Algorithm

- 1) Write a list of numbers from 2 to the largest number you want to test for primality. Call this List A.
- 2) Write the number 2, the first prime number, in another list for primes found. Call this List B.
- 3) Strike off 2 and all multiples of 2 from List A.
- 4) The first remaining number in the list is a prime number. Write this number into List B.
- 5) Strike off this number and all multiples of this number from List A. The crossing-off of multiples can be started at the square of the number, as lower multiples have already been crossed out in previous steps.
- 6) Repeat steps 4 through 6 until no more numbers are left in List A."

To run the demo, execute the entry point function **primesCompute()** from the target shell. This function will create *numTasks* computational tasks to compute prime numbers from 2 to *maxPrimeNum*.

In a VxWorks SMP system, the computational tasks will be assigned to separate CPUs. Of course, if *numTasks* exceeds the number of CPUs in the system, the additional tasks will not contribute to reducing the elapsed computation time. In fact, the additional tasks may contribute to increasing the elapsed computation time when round-robin scheduling is enabled.

The following example output was obtained from a Freescale HPC-NET (8641D dual-core processor) board:

```
-> primesCompute
Usage: primesCompute <unsigned maxPrimeNum>, <unsigned int numTasks>
value = -1 = 0xffffffff

-> primesCompute 10000000, 1
Computing primes... done.

Number of primes numbers found: 664579      Computation time = 131 ticks

The following prime numbers were computed:
```

	2	3	5	7	11	13	17	19	23
29									
	31	37	41	43	47	53	59	61	67
71									
	73	79	83	89	97	101	103	107	109
113									

	127	131	137	139	149	151	157	163	167
173									
	179	181	191	193	197	199	211	223	227
229									
	233	239	241	251	257	263	269	271	277
281									
	283	293	307	311	313	317	331	337	347
349									
	353	359	367	373	379	383	389	397	401
409									
	419	421	431	433	439	443	449	457	461
463									
	467	479	487	491	499	503	509	521	523
541									

Type <CR> to continue, Q<CR> to stop: q

-> primesCompute 10000000, 2
Computing primes... done.

Number of primes numbers found: 664579 Computation time = 113 ticks

The following prime numbers were computed:

	2	3	5	7	11	13	17	19	23
29									
	31	37	41	43	47	53	59	61	67
71									
	73	79	83	89	97	101	103	107	109
113									
	127	131	137	139	149	151	157	163	167
173									
	179	181	191	193	197	199	211	223	227
229									
	233	239	241	251	257	263	269	271	277
281									
	283	293	307	311	313	317	331	337	347
349									
	353	359	367	373	379	383	389	397	401
409									
	419	421	431	433	439	443	449	457	461
463									
	467	479	487	491	499	503	509	521	523
541									

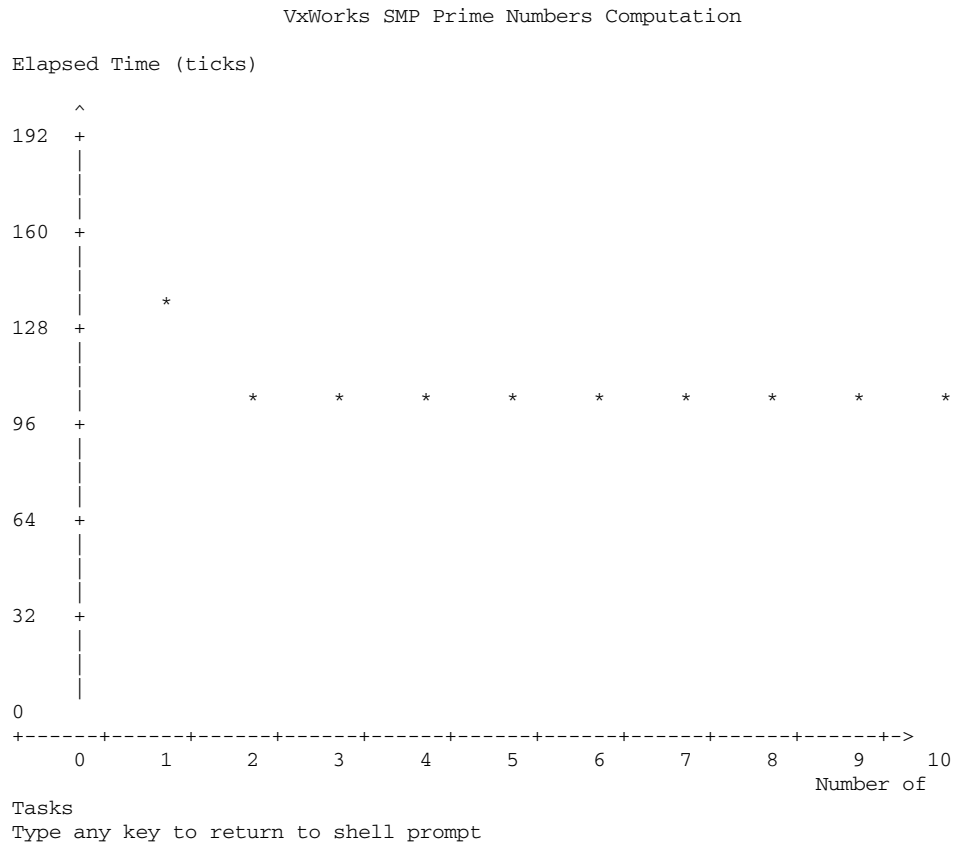
Type <CR> to continue, Q<CR> to stop: q
->

Specifying a *numTasks* of 0 selects "graph" mode. Graph mode will repeatedly compute prime numbers from 2 to *maxPrimeNum* using 1 to *numTasks* computational tasks. The compute times are plotted on an ASCII graph on standard output (STD_OUT). The x-axis represents the number of tasks used to compute prime numbers, and the y-axis represents the elapsed computation time.

ANSI escape sequences are used to print bold/underline text and to perform cursor movements. Thus the console terminal program, e.g. HyperTerminal for PCs, must support ANSI escape sequences, and should be sized with 35 rows (or more), and 80 columns (or more).

The following example output was obtained from a Freescale HPC-NET (8641D dual-core processor) board:

```
-> primesCompute 10000000
```



INCLUDE FILES none

pthreadLib

NAME	pthreadLib – POSIX 1003.1c thread library interfaces
ROUTINES	<p>pthread_sigmask() – change and/or examine calling thread's signal mask (POSIX)</p> <p>pthread_kill() – send a signal to a thread (POSIX)</p> <p>pthread_mutexattr_init() – initialize mutex attributes object (POSIX)</p> <p>pthread_mutexattr_destroy() – destroy mutex attributes object (POSIX)</p> <p>pthread_mutexattr_setprotocol() – set protocol attribute in mutex attribute object (POSIX)</p> <p>pthread_mutexattr_getprotocol() – get value of protocol in mutex attributes object (POSIX)</p> <p>pthread_mutexattr_setprioceiling() – set prioceiling attribute in mutex attributes object (POSIX)</p> <p>pthread_mutexattr_getprioceiling() – get the current value of the prioceiling attribute in a mutex attributes object (POSIX)</p> <p>pthread_mutex_getprioceiling() – get the value of the prioceiling attribute of a mutex (POSIX)</p> <p>pthread_mutex_setprioceiling() – dynamically set the prioceiling attribute of a mutex (POSIX)</p> <p>pthread_mutex_init() – initialize mutex from attributes object (POSIX)</p> <p>pthread_mutex_destroy() – destroy a mutex (POSIX)</p> <p>pthread_mutex_lock() – lock a mutex (POSIX)</p> <p>pthread_mutex_trylock() – lock mutex if it is available (POSIX)</p> <p>pthread_mutex_unlock() – unlock a mutex (POSIX)</p> <p>pthread_condattr_init() – initialize a condition attribute object (POSIX)</p> <p>pthread_condattr_destroy() – destroy a condition attributes object (POSIX)</p> <p>pthread_cond_init() – initialize condition variable (POSIX)</p> <p>pthread_cond_destroy() – destroy a condition variable (POSIX)</p> <p>pthread_cond_signal() – unblock a thread waiting on a condition (POSIX)</p> <p>pthread_cond_broadcast() – unblock all threads waiting on a condition (POSIX)</p> <p>pthread_cond_wait() – wait for a condition variable (POSIX)</p> <p>pthread_cond_timedwait() – wait for a condition variable with a timeout (POSIX)</p> <p>pthread_attr_setscope() – set contention scope for thread attributes (POSIX)</p> <p>pthread_attr_getscope() – get contention scope from thread attributes (POSIX)</p> <p>pthread_attr_setinheritsched() – set inheritsched attribute in thread attribute object (POSIX)</p> <p>pthread_attr_getinheritsched() – get current value if inheritsched attribute in thread attributes object (POSIX)</p> <p>pthread_attr_setschedpolicy() – set schedpolicy attribute in thread attributes object (POSIX)</p> <p>pthread_attr_getschedpolicy() – get schedpolicy attribute from thread attributes object (POSIX)</p> <p>pthread_attr_setschedparam() – set schedparam attribute in thread attributes object (POSIX)</p>

pthread_attr_getschedparam() – get value of schedparam attribute from thread attributes object (POSIX)
pthread_getschedparam() – get value of schedparam attribute from a thread (POSIX)
pthread_setschedparam() – dynamically set schedparam attribute for a thread (POSIX)
pthread_attr_init() – initialize thread attributes object (POSIX)
pthread_attr_destroy() – destroy a thread attributes object (POSIX)
pthread_attr_setopt() – set options in thread attribute object
pthread_attr_getopt() – get options from thread attribute object
pthread_attr_setname() – set name in thread attribute object
pthread_attr_getname() – get name of thread attribute object
pthread_attr_setstacksize() – set stacksize attribute in thread attributes object (POSIX)
pthread_attr_getstacksize() – get stack value of stacksize attribute from thread attributes object (POSIX)
pthread_attr_setstackaddr() – set stackaddr attribute in thread attributes object (POSIX)
pthread_attr_getstackaddr() – get value of stackaddr attribute from thread attributes object (POSIX)
pthread_attr_setdetachstate() – set detachstate attribute in thread attributes object (POSIX)
pthread_attr_getdetachstate() – get value of detachstate attribute from thread attributes object (POSIX)
pthread_create() – create a thread (POSIX)
pthread_detach() – dynamically detach a thread (POSIX)
pthread_join() – wait for a thread to terminate (POSIX)
pthread_exit() – terminate a thread (POSIX)
pthread_equal() – compare thread IDs (POSIX)
pthread_self() – get the calling thread's ID (POSIX)
pthread_once() – dynamic package initialization (POSIX)
pthread_key_create() – create a thread specific data key (POSIX)
pthread_setspecific() – set thread specific data (POSIX)
pthread_getspecific() – get thread specific data (POSIX)
pthread_key_delete() – delete a thread specific data key (POSIX)
pthread_cancel() – cancel execution of a thread (POSIX)
pthread_setcancelstate() – set cancellation state for calling thread (POSIX)
pthread_setcanceltype() – set cancellation type for calling thread (POSIX)
pthread_testcancel() – create a cancellation point in the calling thread (POSIX)
pthread_cleanup_push() – pushes a routine onto the cleanup stack (POSIX)
pthread_cleanup_pop() – pop a cleanup routine off the top of the stack (POSIX)

DESCRIPTION

This library provides an implementation of POSIX 1003.1c threads for VxWorks. This provides an increased level of compatibility between VxWorks applications and those written for other operating systems that support the POSIX threads model (often called *pthread*s).

VxWorks implements POSIX threads in the kernel based on tasks. Because the kernel environment is different from a process environment, in the POSIX sense, there are a few restrictions in the implementation, but in general, since tasks are roughly equivalent to

threads, the *pthreads* support maps well onto VxWorks. The restrictions are explained in more detail in the following paragraphs.

CONFIGURATION	To add POSIX threads support to a system, the component INCLUDE_POSIX_PTHREADS must be added.
THREADS	<p>A thread is essentially a VxWorks task, with some additional characteristics. The first is detachability, where the creator of a thread can optionally block until the thread exits. The second is cancelability, where one task or thread can cause a thread to exit, possibly calling cleanup handlers. The next is private data, where data private to a thread is created, accessed and deleted via keys. Each thread has a unique ID. A thread's ID is different than it's VxWorks task ID.</p> <p>It is recommended to use the POSIX thread API only via POSIX threads, not via native VxWorks tasks. Since pthreads are not created by default in VxWorks the pthread_create() API can be safely used by a native VxWorks task in order to create the first POSIX thread. If a native VxWorks task must use more pthread API it is recommended to give this task a pthread persona by calling pthread_self() first.</p>
MUTEXES	<p>Included with the POSIX threads facility is a mutual exclusion facility, or <i>mutex</i>. These are functionally similar to the VxWorks mutex semaphores (see semMLib for more detail), and in fact are implemented using a VxWorks mutex semaphore. The advantage they offer, like all of the POSIX libraries, is the ability to run software designed for POSIX platforms under VxWorks.</p> <p>There are three types of locking protocols available: PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT and PTHREAD_PRIO_PROTECT. PTHREAD_PRIO_INHERIT, which is the default, maps to a semaphore created with SEM_Q_PRIORITY and SEM_INVERSION_SAFE set (see semMCreate for more detail). A thread locking a mutex created with its protocol attribute set to PTHREAD_PRIO_PROTECT has its priority elevated to that of the prioceiling attribute of the mutex. When the mutex is unlocked, the priority of the calling thread is restored to its previous value. Both protocols aim at solving the priority inversion problem where a lower priority thread can unduly delay a higher priority thread requiring the resource blocked by the lower priority thread. The PTHREAD_PRIO_INHERIT protocol can be more efficient since it elevates the priority of a thread only when needed. The PTHREAD_PRIO_PROTECT protocol gives more control over the priority change at the cost of systematically elevating the thread's priority as well as preventing threads to use a mutex which priority ceiling is lower than the thread's priority. In contrast the PTHREAD_PRIO_NONE protocol does not affect the priority and scheduling of the thread that owns the mutex.</p>
CONDITION VARIABLES	<p>Condition variables are another synchronization mechanism that is included in the POSIX threads library. A condition variable allows threads to block until some condition is met. There are really only two basic operations that a condition variable can be involved in: waiting and signalling. Condition variables are always associated with a mutex.</p>

A thread can wait for a condition to become true by taking the mutex and then calling **pthread_cond_wait()**. That function will release the mutex and wait for the condition to be signalled by another thread. When the condition is signalled, the function will re-acquire the mutex and return to the caller.

Condition variable support two types of signalling: single thread wake-up using **pthread_cond_signal()**, and multiple thread wake-up using **pthread_cond_broadcast()**. The latter of these will unblock all threads that were waiting on the specified condition variable.

It should be noted that condition variable signals are not related to POSIX signals. In fact, they are implemented using VxWorks semaphores.

RESOURCE COMPETITION

All tasks, and therefore all POSIX threads, compete for CPU time together. For that reason the contention scope thread attribute is always **PTHREAD_SCOPE_SYSTEM**.

NO VXWORKS EQUIVALENT

Since there is no notion of a process (in the POSIX sense) in the kernel environment, there is no notion of sharing of locks (mutexes) and condition variables between processes. As a result, the POSIX symbol **_POSIX_THREAD_PROCESS_SHARED** is not defined in this implementation, and the routines **pthread_condattr_getpshared()**, **pthread_condattr_setpshared()**, **pthread_mutexattr_getpshared()** are not implemented.

Also, since the VxWorks kernel is not a process environment, **fork()**, **wait()**, and **pthread_atfork()** are unimplemented.

SCHEDULING

The default scheduling policy for a created thread is inherited from the system setting at the time of creation.

Unlike for the pthread support in RTPs, the POSIX threads in the kernel are not scheduled by the POSIX scheduler. They are scheduled by the VxWorks native scheduler, like all other VxWorks tasks: scheduling policies under VxWorks are global; they are not set per-thread, as the POSIX model describes. As a result, the *pthread* scheduling routines, as well as the POSIX scheduling routines native to VxWorks, do not allow you to change the scheduling policy for kernel pthreads. Under VxWorks you may set the scheduling policy in a thread, but if it does not match the system's scheduling policy, an error is returned.

The detailed explanation for why this situation occurs is a bit convoluted: technically the scheduling policy is an attribute of a thread (in that there are **pthread_attr_getschedpolicy()** and **pthread_attr_setschedpolicy()** functions that define what the thread's scheduling policy will be once it is created, and not what any thread should do at the time they are called). A situation arises where the scheduling policy in force at the time of a thread's creation is not the same as set in its attributes. In this case **pthread_create()** fails with the error **EPERM**.

The bottom line is that under VxWorks, if you wish to specify the scheduling policy of a kernel thread, you must set the desired global scheduling policy to match. Kernel threads

must then adhere to that scheduling policy, or use the **PTHREAD_INHERIT_SCHED** mode to inherit the current mode and creator's priority. Alternatively, you can also use pthreads in an RTP.

In the kernel, the POSIX scheduling policies are therefore mapped as follows:

- SCHED_FIFO**
is mapped on VxWorks' preemptive priority scheduling.
- SCHED_RR**
is mapped on VxWorks' round-robin scheduling.
- SCHED_OTHER**
is mapped on the active VxWorks scheduling policy (either preemptive priority scheduling or round-robin scheduling). This is the only meaningful scheduling policy for kernel pthreads.

CREATION AND CANCELLATION

Each time a thread is created, the *pthreads* library allocates resources on behalf of it. Each time a VxWorks task (i.e. one not created by the **pthread_create()** function) uses a POSIX threads feature such as thread private data or pushes a cleanup handler, the *pthreads* library creates resources on behalf of that task as well.

Asynchronous thread cancellation is accomplished by way of a signal. A special signal, SIGCNCL, has been set aside in this version of VxWorks for this purpose. Applications should take care not to block or handle SIGCNCL.

Current cancellation points in system and library calls:

Libraries	Cancellation Points
aioPxLib	aio_suspend
ioLib	creat, open, read, write, close, fsync, fdatasync, fcntl
mqPxLib	mq_receive, mq_send
pthreadLib	pthread_cond_timedwait, pthread_cond_wait, pthread_join, pthread_testcancel
semPxLib	sem_wait
sigLib	pause, sigsuspend, sigtimedwait, sigwait, sigwaitinfo, waitpid
timerLib	sleep, nanosleep

Caveat: due to the implementation of some of the I/O drivers in VxWorks, it is possible that a thread cancellation request can not actually be honored.

SUMMARY MATRIX

<i>pthread</i> function	Implemented?	Note(s)
pthread_attr_destroy()	Yes	
pthread_attr_getdetachstate()	Yes	
pthread_attr_getinheritsched()	Yes	
pthread_attr_getname()	Yes	6
pthread_attr_getopt()	Yes	6

<i>pthread</i> function	Implemented?	Note(s)
<code>pthread_attr_getschedparam()</code>	Yes	
<code>pthread_attr_getschedpolicy()</code>	Yes	
<code>pthread_attr_getscope()</code>	Yes	
<code>pthread_attr_getstackaddr()</code>	Yes	
<code>pthread_attr_getstacksize()</code>	Yes	
<code>pthread_attr_init()</code>	Yes	
<code>pthread_attr_setdetachstate()</code>	Yes	
<code>pthread_attr_setinheritsched()</code>	Yes	
<code>pthread_attr_setname()</code>	Yes	6
<code>pthread_attr_setopt()</code>	Yes	6
<code>pthread_attr_setschedparam()</code>	Yes	
<code>pthread_attr_setschedpolicy()</code>	Yes	
<code>pthread_attr_setscope()</code>	Yes	2
<code>pthread_attr_setstackaddr()</code>	Yes	
<code>pthread_attr_setstacksize()</code>	Yes	
<code>pthread_atfork()</code>	No	1
<code>pthread_cancel()</code>	Yes	5
<code>pthread_cleanup_pop()</code>	Yes	
<code>pthread_cleanup_push()</code>	Yes	
<code>pthread_condattr_destroy()</code>	Yes	
<code>pthread_condattr_getpshared()</code>	No	3
<code>pthread_condattr_init()</code>	Yes	
<code>pthread_condattr_setpshared()</code>	No	3
<code>pthread_cond_broadcast()</code>	Yes	
<code>pthread_cond_destroy()</code>	Yes	
<code>pthread_cond_init()</code>	Yes	
<code>pthread_cond_signal()</code>	Yes	
<code>pthread_cond_timedwait()</code>	Yes	
<code>pthread_cond_wait()</code>	Yes	
<code>pthread_create()</code>	Yes	
<code>pthread_detach()</code>	Yes	
<code>pthread_equal()</code>	Yes	
<code>pthread_exit()</code>	Yes	
<code>pthread_getschedparam()</code>	Yes	4
<code>pthread_getspecific()</code>	Yes	
<code>pthread_join()</code>	Yes	
<code>pthread_key_create()</code>	Yes	
<code>pthread_key_delete()</code>	Yes	
<code>pthread_kill()</code>	Yes	
<code>pthread_once()</code>	Yes	
<code>pthread_self()</code>	Yes	
<code>pthread_setcancelstate()</code>	Yes	
<code>pthread_setcanceltype()</code>	Yes	
<code>pthread_setschedparam()</code>	Yes	4

<i>pthread</i> function	Implemented?	Note(s)
pthread_setspecific()	Yes	
pthread_sigmask()	Yes	
pthread_testcancel()	Yes	
pthread_mutexattr_destroy()	Yes	
pthread_mutexattr_getprioceiling()	Yes	
pthread_mutexattr_getprotocol()	Yes	
pthread_mutexattr_getpshared()	No	3
pthread_mutexattr_init()	Yes	
pthread_mutexattr_setprioceiling()	Yes	
pthread_mutexattr_setprotocol()	Yes	
pthread_mutexattr_setpshared()	No	3
pthread_mutex_destroy()	Yes	
pthread_mutex_getprioceiling()	Yes	
pthread_mutex_init()	Yes	
pthread_mutex_lock()	Yes	
pthread_mutex_setprioceiling()	Yes	
pthread_mutex_trylock()	Yes	
pthread_mutex_unlock()	Yes	

- NOTES**
- 1 The **pthread_atfork()** function is not implemented since **fork()** is not implemented in VxWorks.
 - 2 The contention scope thread scheduling attribute is always **PTHREAD_SCOPE_SYSTEM**, since threads (i.e. tasks) contend for resources with all other threads in the system.
 - 3 The routines **pthread_condattr_getpshared()**, **pthread_attr_setpshared()**, **pthread_mutexattr_getpshared()** and **pthread_mutexattr_setpshared()** are not supported, since these interfaces describe how condition variables and mutexes relate to a process, and the VxWorks kernel is not a process environment.
 - 4 The default scheduling policy is inherited from the current system setting. The POSIX model of per-thread scheduling policies is not supported, since a basic tenet of the design of VxWorks is a system-wide scheduling policy.
 - 5 Thread cancellation is supported in appropriate *pthread* routines and those routines already supported by VxWorks. However, the complete list of cancellation points specified by POSIX is not supported because routines such as **msync()**, **tcdrain()**, and **wait()** are not implemented by VxWorks.
 - 6 VxWorks-specific routines provided as an extension to IEEE Std 1003.1 in order to handle VxWorks tasks' attributes.

INCLUDE FILES **pthread.h**

SEE ALSO **taskLib, semMLib, semPxLib**

ptyDrv

NAME	ptyDrv – pseudo-terminal driver
ROUTINES	ptyDrv() – initialize the pseudo-terminal driver ptyDevCreate() – create a pseudo terminal ptyDevRemove() – destroy a pseudo terminal
DESCRIPTION	The pseudo-terminal driver provides a tty-like interface between a master and slave process, typically in network applications. The master process simulates the "hardware" side of the driver (e.g., a USART serial chip), while the slave process is the application program that normally talks to the driver.
CONFIGURATION	To use the pseudo-terminal driver library, configure VxWorks with the INCLUDE_PTYDRV component.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, the following routines must be called directly: **ptyDrv()** to initialize the driver, **ptyDevCreate()** to create devices, and **ptyDevRemove()** to remove an existing device.

INITIALIZING THE DRIVER

Before using the driver, it must be initialized by calling **ptyDrv()**. This routine must be called before any reads, writes, or calls to **ptyDevCreate()**.

CREATING PSEUDO-TERMINAL DEVICES

Before a pseudo-terminal can be used, it must be created by calling **ptyDevCreate()**:

```
STATUS ptyDevCreate
(
    char *name,          /* name of pseudo terminal */
    int  rdBufSize,      /* size of terminal read buffer */
    int  wrtBufSize      /* size of write buffer */
)
```

For instance, to create the device pair **"/pty0.M"** and **"/pty0.S"**, with read and write buffer sizes of 512 bytes, the proper call would be:

```
ptyDevCreate ("/pty0.", 512, 512);
```

When **ptyDevCreate()** is called, two devices are created, a master and slave. One is called *nameM* and the other *nameS*. They can then be opened by the master and slave processes. Data written to the master device can then be read on the slave device, and vice versa. Calls to **ioctl()** may be made to either device, but they should only apply to the slave side, since the master and slave are the same device.

The **ptyDevRemove()** routine will delete an existing pseudo-terminal device and reclaim the associated memory. Any file descriptors associated with the device will be closed.

IOCTL FUNCTIONS	Pseudo-terminal drivers respond to the same ioctl() functions used by <i>tty</i> devices. These functions are defined in ioLib.h and documented in the manual entry for tyLib .
INCLUDE FILES	ioLib.h , ptyDrv.h
SEE ALSO	tyLib , the VxWorks programmer guides.

quiccEngineUtils

NAME	quiccEngineUtils – quciic engine resource allocation
ROUTINES	quiccEngineRegister() – register quiccEngine driver quiccEngineDrvCtrlShow() – place holder just prints out control structure ptr
DESCRIPTION	Utilities to supply access to shared Quicc Engine resources to device drivers.
INCLUDE FILES	none

rBuffLib

NAME	rBuffLib – dynamic ring buffer (rBuff) library
ROUTINES	wwrBuffMgrPrioritySet() – set the priority of the System Viewer rBuff manager
DESCRIPTION	This library contains a routine for changing the default priority of the rBuff manager task.
INCLUDE FILES	none
SEE ALSO	memLib , rngLib , <i>"VxWorks Kernel Programmer's Guide: Basic OS"</i>

ramDiskCbio

NAME	ramDiskCbio – RAM Disk Cached Block Driver
-------------	---

ROUTINES	ramDiskDevCreate() – Initialize a RAM Disk device
DESCRIPTION	<p>This module implements a RAM-disk driver with a CBIO interface which can be directly utilized by dosFsLib without the use of the Disk Cache module dcacheCbio. This results in an ultra-compact RAM footprint. This module is implemented using the CBIO API (see cbioLib())</p> <p>This module is delivered in source as a functional example of a basic CBIO module.</p>
CAVEAT	This module may be used for SRAM or other non-volatile RAM cards to store a file system, but that configuration will be susceptible to data corruption in events of system failure which are not normally observed with magnetic disks, i.e. using this driver with an SRAM card can not guard against interruptions in midst of updating a particular sector, resulting in that sector become internally inconsistent.
INCLUDE FILES	none
SEE ALSO	dosFsLib , cbioLib

ramDrv

NAME	ramDrv – RAM disk driver
ROUTINES	ramDrv() – prepare a RAM disk driver for use (optional) ramDevCreate() – create a RAM disk device

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, can be called directly by the user. The first, **ramDrv()**, provides no real function except to parallel the initialization function found in true disk device drivers. A call to **ramDrv()** is not required to use the RAM disk driver. However, the second routine, **ramDevCreate()**, must be called directly to create RAM disk devices.

Once the device has been created, it must be associated with a name and file system (**dosFs**, **hrfs**, or **rawFs**). This is accomplished in a two-stage process. First, create an XBD wrapper around the pointer to the block device structure returned by **ramDevCreate()**. Second, format the drive with the desired file system.

```
BLK_DEV * pBlkDev;

pBlkDev = ramDevCreate (NULL, 512, 32, 416, 0);

xbdBlkDevCreate (pBlkDev, "/ramDrv");

dosFsVolFormat ("/ramDrv:0", DOS_OPT_BLANK, NULL);
```

See the reference entry for **ramDevCreate()** for a more detailed discussion.

IOCTL FUNCTIONS

The RAM driver is called in response to **ioctl()** codes in the same manner as a normal disk driver. When the file system is unable to handle a specific **ioctl()** request, it is passed to the **ramDrv** driver. Although there is no physical device to be controlled, **ramDrv** does handle a FIODISKFORMAT request, which always returns OK. All other **ioctl()** requests return an error and set the task's **errno** to **S_ioLib_UNKNOWN_REQUEST**.

INCLUDE FILE **ramDrv.h**

SEE ALSO **xbdBlkDevCreate()**, **dosFsVolFormat()**, and **hrfsFormat()**, the VxWorks programmer guides.

rawFsLib

NAME **rawFsLib** – raw block device file system library

ROUTINES **rawFsDevInit()** – associate a block device with raw volume functions
 rawFsInit() – prepare to use the raw volume library

USING THIS LIBRARY

The various routines provided by the VxWorks raw "file system" (**rawFs**) may be separated into three broad groups: general initialization, device initialization, and file system operation.

The **rawFsInit()** routine is the principal initialization function; it need only be called once, regardless of how many **rawFs** devices will be used.

A separate **rawFs** routine is used for device initialization. For each **rawFs** device, **rawFsDevInit()** must be called to install the device.

INITIALIZATION Before any other routines in **rawFsLib** can be used, **rawFsInit()** must be called to initialize the library. This call specifies the maximum number of raw device file descriptors that can be open simultaneously and allocates memory for that many raw file descriptors. Any attempt to open more raw device file descriptors than the specified maximum will result in errors from **open()** or **creat()**.

During the **rawFsInit()** call, the raw device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, **rawFsDrvNum**.

This initialization is enabled when the configuration macro **INCLUDE_RAWFS** is defined; **rawFsInit()** is then called (normally via components from **vxprj**).

DEFINING A RAW DEVICE

To use this library for a particular device, and eXtended Block Device (XBD) must be initialized as the backing media; the second parameter to **rawFsDevInit()** must be a `device_t` referring to this XBD.

The **rawFsDevInit()** routine is used to associate a device with the **rawFsLib** functions. The *pVolName* parameter expected by **rawFsDevInit()** is a pointer to a name string, to be used to identify the device. This will serve as the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using **iosDevShow()**.

The syntax of the **rawFsDevInit()** routine is as follows:

```
rawFsDevInit
(
    char      *pVolName, /* name to be used for volume - iosDevAdd */
    device_t xbd        /* handle to the backing XBD */
)
```

When **rawFsLib** receives a request from the I/O system, after **rawFsDevInit()** has been called, it calls the appropriate device driver routines to access the device.

IOCTL FUNCTIONS

The VxWorks raw block device file system supports the following **ioctl()** functions. The functions listed are defined in the header **ioLib.h**.

FIOGETNAME

Gets the file name of the file descriptor and copies it to the buffer *nameBuf*:

```
status = ioctl (fd, FIOGETNAME, &nameBuf);
```

FIOSEEK

Sets the current byte offset on the disk to the position specified by *newOffset*:

```
status = ioctl (fd, FIOSEEK, newOffset);
```

FIOWHERE

Returns the current byte position from the start of the device for the specified file descriptor. This is the byte offset of the next byte to be read or written. It takes no additional argument:

```
position = ioctl (fd, FIOWHERE, 0);
```

FIOFLUSH

Writes all modified file descriptor buffers to the physical device.

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOSYNC

Performs the same function as FIOFLUSH.

FIONREAD

Copies to *unreadCount* the number of bytes from the current file position to the end of the device:

```
status = ioctl (fd, FIONREAD, &unreadCount);
```

INCLUDE FILES **rawFsLib.h**

SEE ALSO **ioLib**, **iosLib**, **rawFsLib**, **ramDrv**, the VxWorks programmer guides.

rawPerfDemo

NAME **rawPerfDemo** – VxWorks/SMP raw performance demo

ROUTINES **rawPerfDemo()** – entry point for the VxWorks/SMP raw performance demo

DESCRIPTION This module implements a VxWorks/SMP raw performance demonstration. The term "raw performance" is used to indicate that the performance figures generated by this demo are a result of a number crunching algorithm, as opposed to an I/O intensive and/or kernel system call intensive application. In addition, each CPU operates on per-CPU data, i.e. the CPUs in the system do not cooperate to achieve a common goal.

The numerical computation performed by each CPU is a simple formula to compute pi (ratio of the circumference to the diameter of a circle) using floating-point arithmetic. Aggregate raw performance is reported in "operations per second"; an operation consisting of computing pi to a certain number of decimal places. The actual performance figures, i.e. the operations per second, are not important for the purposes of this demo. Instead, the relative improvement in performance that occurs as additional CPUs are "enabled" is important. Generally, the aggregate raw performance of a VxWorks/SMP system will increase linearly with the number of CPUs enabled.

Aggregate raw performance figures are first obtained with only a single CPU enabled (actually the other CPUs are enabled but are idle). Performance figures are obtained for a period of 5 seconds in 1-second increments. After the initial 5-second period completes, an additional CPU is added to the performance run. Performance figures are again obtained for another 5 seconds as described above. This process is continued until all CPUs in the system have been added to the performance run.

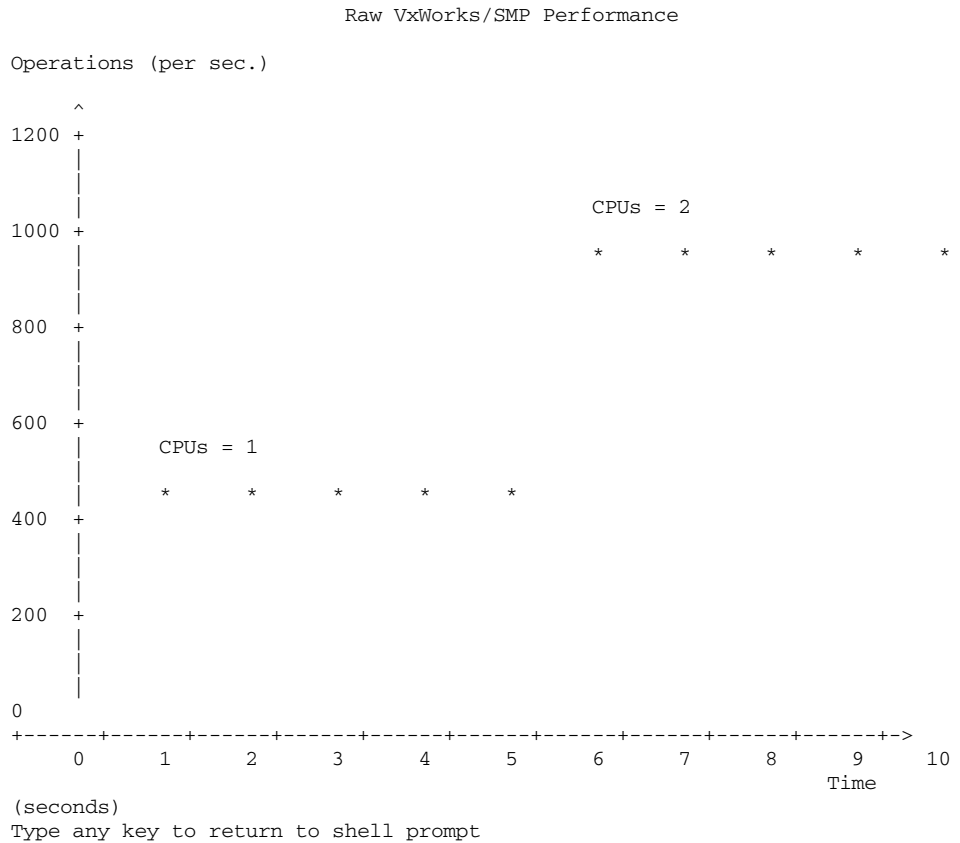
The aggregate raw performance results are displayed as an ASCII character graph on standard output (**STD_OUT**). The elapsed time is plotted on the x-axis and "operations per second" is plotted on the y-axis. The data is plotted in real-time, i.e. the data points are plotted as aggregate raw performance figures are obtained.

ANSI escape sequences are used to print bold/underline text and to perform cursor movements. Thus the console terminal program, e.g. HyperTerminal for PCs, must support ANSI escape sequences, and should be sized with 35 rows (or more), and 80 columns (or more).

As mentioned above, generally, the "raw performance" of a VxWorks/SMP system will increase linearly with the number of CPUs enabled. Thus, the displayed graph will resemble a step function; a step in aggregate raw performance will occur as each CPU is enabled.

The following example output was obtained from a Freescale HPC-NET (8641D dual-core processor) board:

```
-> rawPerfDemo
```



INCLUDE FILES none

rebootLib

NAME	rebootLib – reboot support library
ROUTINES	reboot() – reset network devices and transfer control to boot ROMs rebootHookAdd() – add a routine to be called at reboot
DESCRIPTION	This library provides reboot support. To restart VxWorks, the routine reboot() can be called at any time by typing CTRL-X from the shell. Shutdown routines can be added with rebootHookAdd() . These are typically used to reset or synchronize hardware. For example, netLib adds a reboot hook to cause all network interfaces to be reset. Once the reboot hooks have been run, sysToMonitor() is called to transfer control to the boot ROMs. For more information, see the reference entry for bootInit .
CONFIGURATION	The reboot support library is always included in VxWorks.
DEFICIENCIES	<p>The order in which hooks are added is the order in which they are run. As a result, netLib will kill the network, and no user-added hook routines will be able to use the network. There is no rebootHookDelete() routine.</p> <p>Reboot hooks must not invoke kernel service routines that may block. Blocking calls within the reboot hooks may cause the reboot process to reschedule() or hang, potentially leaving the system in an unpredictable state.</p>
INCLUDE FILES	rebootLib.h
SEE ALSO	sysLib , bootConfig , bootInit

rngLib

NAME	rngLib – ring buffer subroutine library
ROUTINES	rngCreate() – create an empty ring buffer rngDelete() – delete a ring buffer rngFlush() – make a ring buffer empty rngBufGet() – get characters from a ring buffer rngBufPut() – put bytes into a ring buffer rngIsEmpty() – test if a ring buffer is empty rngIsFull() – test if a ring buffer is full (no more room) rngFreeBytes() – determine the number of free bytes in a ring buffer rngNBytes() – determine the number of bytes in a ring buffer

rngPutAhead() – put a byte ahead in a ring buffer without moving ring pointers
rngMoveAhead() – advance a ring pointer by *n* bytes

DESCRIPTION

This library provides routines for creating and using ring buffers, which are first-in-first-out circular buffers. The routines simply manipulate the ring buffer data structure; no kernel functions are invoked. In particular, ring buffers by themselves provide no task synchronization or mutual exclusion.

However, the ring buffer pointers are manipulated in such a way that a reader task (invoking **rngBufGet()**) and a writer task (invoking **rngBufPut()**) can access a ring simultaneously without requiring mutual exclusion. This is because readers only affect a *read* pointer and writers only affect a *write* pointer in a ring buffer data structure. However, access by multiple readers or writers *must* be interlocked through a mutual exclusion mechanism (i.e., a mutual-exclusion semaphore guarding a ring buffer).

This library also supplies two macros, **RNG_ELEM_PUT** and **RNG_ELEM_GET**, for putting and getting single bytes from a ring buffer. They are defined in **rngLib.h**.

```
int RNG_ELEM_GET (ringId, pch, fromP)
int RNG_ELEM_PUT (ringId, ch, toP)
```

Both macros require a temporary variable *fromP* or *toP*, which should be declared as **register int** for maximum efficiency. **RNG_ELEM_GET** returns 1 if there was a character available in the buffer; it returns 0 otherwise. **RNG_ELEM_PUT** returns 1 if there was room in the buffer; it returns 0 otherwise. These are somewhat faster than **rngBufPut()** and **rngBufGet()**, which can put and get multi-byte buffers.

INCLUDE FILES

rngLib.h

rtl8139VxbEnd

NAME

rtl8139VxbEnd – RealTek 8139/8100 10/100 VxBus END driver

ROUTINES

rtlRegister() – register with the VxBus subsystem

DESCRIPTION

This module implements a driver for the RealTek 8139 family of PCI 10/100 ethernet controllers. The 8139 family is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The controller has an embedded 10/100 PHY, with a pseudo-MII management interface.

The host communicates with the RealTek 8139 through a single register region, which can be accessed using either I/O space or through a shared memory mapping, and a single PCI interrupt line. the 8139 is a PCI bus master device, however it does not use the traditional descriptor ring model for DMA.

Packet reception is accomplished through a single RX window region, which is mapped into the host's address space. This window can be 8K, 16K, 32K or 64K in size, and must be contiguous. When a packet arrives, a 32-bit RX status word is copied into the start of the RX window, followed by the packet data. The host must examine the RX status word (which contains the frame length) and copy the packet data into an mBlk tuple before handing it off to the stack. The next packet will be copied to the window immediately after the previous one. When the controller reaches the end of the window, it will wrap back to the beginning. A producer index register is used to tell the host how much valid packet data is waiting in the window, and a consumer index register is used by the host to tell the controller how much of the data in the window has been processed.

For transmission, the 8139 has four TX registers pairs. Each pair consists of a data pointer register and a status register. Since there's only one data pointer, outbound packets must reside in a single contiguous buffer. The host copies the address of the packet buffer into the data pointer register and then writes to the status register to initiate transmission. The TX register pairs must be used in sequence: they cannot be used in arbitrary order.

The 8139 receiver supports a 64-bit multicast hash filter, a single unicast address filter, and promiscuous mode.

The 8139 family has a built-in 10/100 PHY, which is accessed via four shortcut registers that provide an MII-compliant management interface. This driver provides miiBus-compliant methods to access the PHY, which allows it to work with the genericPhy driver. This allows the driver to support MII-based IFMEDIA functionality.

BOARD LAYOUT The RealTek 81xx family comprises both PCI and cardbus controllers. The PCI devices are available in both standalone PCI card format and integrated directly onto the system main board. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **rtlRegister()** function, which registers the driver with VxBus.

INCLUDE FILES **rtl8139VxbEnd.h endLib.h netBufLib.h muxLib.h**

SEE ALSO vxBus, **ifLib**, *RealTek 8139 Programming Manual*, <http://www.realtek.com.tw>

rtl8169VxbEnd

NAME **rtl8169VxbEnd** – RealTek 8139C+/8101E/816x/811x VxBus Ethernet driver

ROUTINES **rtgRegister()** – register with the VxBus subsystem

DESCRIPTION

This module implements a driver for the RealTek C+ family of PCI 10/100 and 10/100/1000 ethernet controllers. The C+ family is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The original 8169 chip required an external GMII-compliant PHY, however the 8139C+ and all subsequent chip revs have a copper transceiver built in.

Unlike the original 8139 family, the C+ series of controllers use a standard descriptor-based DMA scheme for host/device data transfer. The first RealTek device to use this mechanism was the 8139C+, which supported both the original 8139 DMA API and the new API. The 8169 and all later devices use the descriptor-based mechanism only.

There are a couple of minor differences between the original 8139C+ and the 8169 and later devices: the 8139C+ only allows a maximum of 64 descriptors per DMA ring, has a couple of its registers at different offsets, and uses the original 8139 PHY register access scheme (the later devices use a single PHY access register). The length fields in the 8139C+'s DMA descriptors are also slightly smaller, since it does not handle jumbo frames. All other aspects of the device API are otherwise identical to the 8169 and later devices.

All devices in the C+ family support TCP/IP checksum offload (for IPv4), hardware VLAN tag stripping and insertion and TCP large send. The gigE devices also support jumbo frames, but only up to 7.5Kb in size. This driver makes use of the checksum offload and VLAN tagging/stripping features.

The following is a list of devices supported by this driver:

RealTek 8139C+ 10/100 PCI RealTek 8169 10/100/1000 PCI (first rev, external PHY) RealTek 8169S/8110S 10/100/1000 PCI (integrated PHY) RealTek 8169SB/8110SB 10/100/1000 PCI (integrated PHY) RealTek 8169SC/8110SC 10/100/1000 PCI (integrated PHY) RealTek 8168B/8111B 10/100/1000 PCIe (integrated PHY) RealTek 8101E 10/100 PCIe (integrated PHY)

BOARD LAYOUT

The RealTek 81xx family comprises PCI, PCIe and cardbus controllers. The PCI devices are available in both standalone PCI card format and integrated directly onto the system main board. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **rtgRegister()** function, which registers the driver with VxBus.

The RealTek gigE devices also support jumbo frames. Note however that the maximum MTU possible is 7400 bytes (not 9000, which is normal for most jumbo-capable NICs). This driver supports jumbo frames on the 8169S/8110S, 8169SB/8110SB, 8169SC/8110SC and 8168B/8111B devices. They are not supported on the 8139C+, 8100E and 8101E devices, which are 10/100 only, nor on the original 8169 (MAC only, no internal PHY) which doesn't seem to be jumbo-capable. (The first generation 8169 is no longer in production however, so this should not be a problem for new designs.)

Jumbo frame support is disabled by default in order to conserve memory (jumbo frames require the use of an buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For

example, to enable jumbo frame support for interface yn0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "rtg", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

INCLUDE FILES **rtl8139VxbEnd.h end.h endLib.h netBufLib.h muxLib.h**

SEE ALSO vxBus, **ifLib**, *RealTek 8139C+ Programming Manual*, <http://www.realtek.com.tw>, *RealTek 8169S/8110S Programming Manual*, <http://www.realtek.com.tw>, *RealTek 8168S/8111S Programming Manual*, <http://www.realtek.com.tw>, *RealTek 8101E Programming Manual*, <http://www.realtek.com.tw>

rtpHookLib

NAME **rtpHookLib** – RTP Hook Support library

ROUTINES **rtpPreCreateHookAdd()** – add a routine to be called before RTP creation.
 rtpPreCreateHookDelete() – delete a previously added RTP pre-create hook.
 rtpPostCreateHookAdd() – add a routine to be called just after RTP creation.
 rtpPostCreateHookDelete() – delete a previously added RTP post-create hook.
 rtpInitCompleteHookAdd() – Add routine to be called after RTP init-complete.
 rtpInitCompleteHookDelete() – delete a previously added RTP init-complete hook
 rtpDeleteHookAdd() – add a routine to be called when RTPs are deleted
 rtpDeleteHookDelete() – delete a previously added RTP delete hook routine

DESCRIPTION This library provides routines for adding extensions to the VxWorks Real-Time Process (RTP) library. This library allows RTP-related facilities to be added to the system without modifying the kernel. The kernel provides call-outs whenever RTP's are created and deleted. These call-outs allow additional routines, or "hooks," to be invoked whenever these events occur. The hook management routines below allow hooks to be dynamically added to and deleted from the current lists of create and delete hooks:

rtpPreCreateHookAdd() and **rtpPreCreateHookDelete()**

Add and delete routines to be called before an RTP is created.

rtpPostCreateHookAdd() and **rtpPostCreateHookDelete()**

Add and delete routines to be called after an RTP and its initial task are created, but before that RTP starts executing.

rtpInitCompleteHookAdd() and **rtpInitCompleteHookDelete()**

Add and delete routines to be called after an RTP is fully created, loaded, initialized, and about to transition to user mode.

rtpDeleteHookAdd() and rtpDeleteHookDelete()

Add and delete routines to be called when an RTP is deleted.

CONFIGURATION To use the RTP hook support library, configure VxWorks with the `INCLUDE_RTP_HOOKS` component.

NOTE It is possible to have dependencies among hook routines. For example, a delete hook may use facilities that are cleaned up and deleted by another delete hook. In such cases, the order in which the hooks run is important. VxWorks runs RTP create hooks in the order in which they were added, and runs RTP delete hooks in reverse of the order in which they were added. Thus, if the hooks are added in "hierarchical" order, such that they rely only on facilities whose hook routines have already been added, then the required facilities will be initialized before any other facilities need them, and will be deleted after all facilities are finished with them.

Typically, creation hooks should be called with *addToHead* set to `FALSE`, and delete hooks should be called to *addToHead* set to `TRUE`.

VxWorks facilities guarantee this by having each facility's initialization routine first call any prerequisite facility's initialization routine before adding its own hooks. Thus, the hooks are always added in the correct order. Each initialization routine protects itself from multiple invocations, allowing only the first invocation to have any effect.

INCLUDE FILES `private/rtpLibP.h`

SEE ALSO `rtpLib`., the VxWorks programmer guides.

rtpLib

NAME `rtpLib` – Real Time Process library

ROUTINES `rtpSpawn()` – spawns a new Real Time Process (RTP) in the system
`rtpDelete()` – terminates a real time process (RTP)

DESCRIPTION This library provides the interfaces to the Real Time Process (RTP) feature. Real Time Process is an optional feature of the VxWorks kernel that provides a process-like environment for applications. In the RTP environment, applications are protected and isolated from each other.

The Real Time Process feature offers the following types of protection:

- protection of the kernel from errant application code
- run-time isolation of applications from each other

- text and read-only data protection
- automatic resource reclamation
- NULL pointer access detection

An RTP is an active entity that always contains active tasks. An RTP may not exist without tasks.

ENABLING RTP SUPPORT

To enable RTP support, the component, **INCLUDE_RTP**, must be added to the kernel at configuration time. This component includes all the functionalities contained in this library and all facilities necessary to support RTP.

To enable monitoring of RTPs, the component, **INCLUDE_RTP_SHOW**, must be configured in conjunction with **INCLUDE_RTP**.

CONFIGURATION RTPs can be configured at creation time via **rtpSpawn()**'s parameters as explained later in this manual and in **rtpSpawn()**'s manual. It is also possible to change the default configuration parameters when the VxWorks image is generated (using Workbench's kernel configuration utility, or the vxprj command line utility). The new default values apply then to all RTPs. These configuration parameters, described in the component description file 01rtp.cdf, are:

RTP_KERNEL_STACK_SIZE

Size of the kernel stack for user tasks.

KERNEL_HEAP_SIZE

Size of the heap reserved to the kernel when RTPs are used in the system.

RTP_HOOK_TBL_SIZE

Number of entries in the RTP create/delete hook tables.

SYSCALL_HOOK_TBL_SIZE

Number of entries in the system call hook tables.

RTP_HEAP_INIT_SIZE

Initial size of the RTP's heap. This can be overridden by the environment variable **HEAP_INITIAL_SIZE**.

RTP_SIGNAL_QUEUE_SIZE

Maximum number of queued signal for a RTP. Note that POSIX requires that this number be at least 32.

RTP CREATION Real Time Processes are created using the **rtpSpawn()** API.

```
rtpSpawn (const char *rtpFileName, const char *argv[], const char
*envp[],
         int priority, int uStackSize, int options, int taskOptions);
```

All RTPs are named and the names are associated with the *rtpFileName* argument passed to the **rtpSpawn()** API.

All RTPs are created with an initial task which is also named after the *rtpFileName* argument passed to the **rtpSpawn()** API: "*iFilename*", where *Filename* is made of the first 30 letters of the file name, excluding the extension.

The creation of an RTP will allocate the necessary memory to load the executable file for the application as well as for the stack of the initial task. Memory for the application is allocated from the global address space and is unique in the system. The memory of an RTP is not static; additional memory may be allocated from the system dynamically after the RTP has been created.

When a RTP is spawned by a kernel task it does not inherit the file descriptors available to this task, except for its task stdin, stdout and stderr file descriptors (0, 1 and 2). However when the RTP is created by another RTP, the child RTP does inherit all file descriptors of its parent.

The environment variables are not inherited from the caller. If the application is expecting specific environment variables, an environment array must be created and passed to the **rtpSpawn()** API. If all of the caller's environment variables must be passed to the RTP, the **envGet()** routine can be used for this purpose (see example below).

The initial task starts its life as a task executing kernel code in supervisor mode. Once the application's code is loaded, the initial task switches to user mode and begins the execution of the application starting at the **_start()** routine (ELF executable's entry point). The initial task initializes the user libraries and invokes all constructors in the application before executing the application's user code. The first user routine in the application is the **main()** function and this function is called after all initializers and constructors are called. All C or C++ applications must provide a **main()** routine. Its complete prototype is as follows:

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

Note that, by convention, only the first two parameters are compulsory:

```
int main
(
    int      argc,    // number of arguments
    char *   argv[]   // NULL terminated array of arguments
)
```

There are attributes that may be set to customize the behavior of the RTP during **rtpSpawn()** (including for example, whether symbol information is to be loaded, the initial task should be stopped at the entry point, or the priority and task options of the initial task.) The manual entry for **rtpSpawn()** provides more details on the options and other configuration parameters available when creating an RTP.

RTP TERMINATION

Real Time Process are terminated in several ways:

- Calling **exit()** within the RTP. This includes the initial task of the RTP reaching the end of its execution.
- When the last task of the RTP exits.
- A fatal **kill()** signal is sent to an RTP.
- An unrecoverable exception occurs

The termination of an RTP will delete the RTP executable and return all memory (virtual and physical memory) used by it to the system. System objects allocated and owned by the RTP will also be deleted from the system. (See **objLib** manual entry for more details on object resource reclamation.) Memory mapped to the RTP will also be freed back into the system. Note that public objects still in use by other users in the system will be inherited by the kernel, and will not be reclaimed at this point.

Any routines registered with the **atexit()** function will be called in the reverse order that they are registered. These **atexit()** routines will be called in a normal termination of an RTP. Abnormal termination of an RTP, such as invoking the deletion from the kernel or sending a fatal **kill()** signal to an RTP, will not cause the **atexit()** routines to be called.

RTP INITIALIZATION

Real Time Processes (RTPs) may be initialized in various ways: automatically by the system during boot time using the RTP startup facility, by launching them from the shell(s), or programmatically using the **rtpSpawn()** API. The automatic initialization is available in four forms:

- Using the **INCLUDE_RTP_APPL_USER** component that enables users to write their own code to spawn their RTPs and to pass parameters to the RTP.
- Using the startup script (s field) in the boot parameters. Users may overload the startup script field to specify RTPs and their parameters to be called at system boot time. The format to use is the following:

```
startup script (s): #print.vxe^hello
```

One or more RTPs may be set up in the startup script field. The # character is the delimiter for each RTP and the ^ is the delimiter for the parameters of the RTP. White spaces are not allowed on the s-field. A sample usage for two RTPs is:

```
startup script (s): #helloworld.vxe#cal.vxe^2004
```

The above line launches helloworld.vxe first, followed by cal.vxe with one argument, being "2004". Additionally, users can also pass options to **rtpSpawn()** itself. The following four options are currently supported:

%p - specifies the priority of the initial task. %s - specifies the execution stack size for the initial task. %o - specifies the value of the *options* argument passed to **rtpSpawn()**. %t - specifies the value of the task options for the RTPs initial task.

The option values may be either in decimal or hexadecimal, in which case they must be prefaced by a "0x" or "0X". Options other than the above four are ignored, and the

RTP is launched with default parameters i.e. with an initial task priority of 220, with a stack size of 64KB, with RTP options being 0, and the initial task having the option **VX_DEALLOC_STACK** being set.

Here is an example usage of the options -

```
startup script (s): #helloworld.vxe^%p=125^%s=0x4000#cal.vxe^2004^%p=150
```

- Using the **INCLUDE_RTP_APPL_INIT_STRING** component that enables users to specify the list of RTPs in the form of a string. The **RTP_APPL_INIT_STRING** parameter must be defined as a quoted string containing the list of RTPs to launch. This component is almost the same as **INCLUDE_RTP_APPL_INIT_BOOTLINE** in both format and behaviour. The difference between the two being that **INCLUDE_RTP_APPL_INIT_STRING** can accept both whitespace and ^ delimiters in its input string.
- Using the **INCLUDE_RTP_APPL_INIT_CMD_SHELL_SCRIPT** component. This allows users to write a shell script using the VxWorks Command Shell syntax, and have it executed after the system boots. The **RTP_APPL_CMD_SCRIPT_FILE** parameter points to a file containing the command shell script to execute. Note that this command shell script is different from the startup script specified in the boot parameters. The latter is a C-interpreter script.

RTPs may be spawned and initialized from the shell(s):

- Using the traditional C interpreter: the **rtpSp()** command will allow the user to execute a VxWorks executable file and pass arguments to its **main()** routine.

```
-> rtpSp "myVxApp.vxe first second third"
```

- Using the RTP command shell by either directly typing the path and name of the executable file and then the list of arguments (similar to a UNIX shell) or use the **rtp exec** command. **help rtp** on the command shell will provide more details.

```
[vxWorks *]# /home/myVxApp.vxe first second third
```

OR

```
[vxWorks *]# rtp exec /home/myVxApp.vxe first second third
```

- Programmatically, from a kernel task or an other RTP, using the **rtpSpawn()** API:

```
const char * args[] = {"/romfs/myApp.vxe", "-arg1", "-arg2 0x1000",
NULL};
```

```
...
```

```
rtpSpawn (args[0], args, NULL, 100, 0x10000, 0, VX_FP_TASK);
```

or (when the caller's environment variables must be passed to the application):

```
rtpSpawn (args[0], args, envGet(0), 100, 0x10000, 0, VX_FP_TASK);
```

Note that the **envGet()** API is available in the kernel space only. In a RTP the *environ* variable is to be used instead. Note also that a specific set of environment variables can be programmatically passed to a RTP via its *envp* parameter:

```
const char * envp[] = {"MY_ENV_VAR1=foo", "MY_ENV_VAR2=bar", NULL};  
...  
rtpSpawn (args[0], args, envp, 100, 0x10000, 0, VX_FP_TASK);
```

TASKS

Every task in the system will have an owner, whether it is the kernel or an RTP. This owner is also the owner of the task object (tasks are <WIND objects>). Unlike other objects, the ownership of a task is restricted to the task's RTP or the kernel. This restriction exists since the task's stack will be allocated from the RTP's memory resources.

By default, tasks running outside the kernel run in the CPU's *user* mode. A task will run in the CPU's *supervisor* mode (**VX_SUPERVISOR_MODE** option is set for the task), if the task is created in the kernel.

The scheduling of tasks is not connected in any way with the RTP that owns them. Even when RTPs are configured into the operating system, tasks are still scheduled based on their priorities and readiness to execute. Note that in the specific case when POSIX threads are executed in the RTP it is mandatory that the POSIX scheduler be used in the system (**INCLUDE_POSIX_PTHREAD_SCHEDULER** component).

Unlike kernel tasks, user tasks (i.e. tasks created in the RTP) cannot have their own private environment variables. They all share the RTP's environment.

Note also that the initial task of a RTP cannot be restarted (see **taskRestart()** for details).

SHARING DATA

The real time process model also supports the sharing of data between RTPs. This sharing can be done using shared data regions. Refer to the **sdLib** manual entries for more information on shared data regions.

To simply share memory, or memory-mapped I/O, with another RTP, a shared data region needs to be created. Then, the *client* RTP (i.e. the one wishing to access the shared resource) simply needs to map the shared data region into its memory space. This is achieved using the **sdMap()** function. See the manual entry for the **sdMap()** function for more information about creating shared data mappings. This sharing relationship must be created at run-time by the application.

SHARING CODE

Sharing of code between RTPs are done using shared libraries. Shared libraries are dynamically loaded at runtime by the RTPs that reference them.

To use shared libraries, the RTP executable must specify at build time that it wants to resolve its undefined symbols using shared libraries. The location of the shared libraries must be provided to the RTP executable using one of the following:

- the **-rpath** *path* compiler flag
- setting the environment variable **LD_LIBRARY_PATH** for the RTP

If the above two options are not used, the location of the RTP executable will be used to find the shared libraries.

Refer to the VxWorks programmer guides for detailed information on how to use shared libraries.

RTP STATES

An RTP life cycle revolves around the following states:

RTP_STATE_CREATE

When an RTP object is created it's initial state is **RTP_STATE_CREATE**. It remains in the state until the RTP object is fully initialized, the image loaded into RTP memory space and the initial task is about to transition to user mode. If initialization is successful, the state transitions to **RTP_STATE_NORMAL** otherwise it transitions to **RTP_STATE_DELETE**.

RTP_STATE_NORMAL

This is the state that indicates that the RTP image is fully loaded and tasks are running in user mode. When the RTP terminates it transitions to **RTP_STATE_DELETE**.

RTP_STATE_DELETE

This is the state that indicates that the RTP is being deleted. No further operations can be performed on the RTP in this state. Once the deletion is complete, the RTP object and its resources are reclaimed by the kernel.

All RTP operations can be done only when the RTP is in **RTP_STATE_CREATE** or **RTP_STATE_NORMAL** state.

RTP STATUS

RTP status bits indicates some important events happening in the RTP life cycle:

RTP_STATUS_STOP

This status bit is set when a stop signal is sent to the RTP. All tasks within the RTP are stopped. A SIGCONT signal sent to the stopped RTP resumes all stopped tasks within the RTP, thus unsetting this bit.

RTP_STATUS_ELECTED_DELETER

This status bit is set once a task is selected to delete the RTP among competing deleting tasks. The RTP is now destined to die. The RTP delete hooks are called after this election, but before the RTP state goes to **RTP_STATE_DELETE**. Once the RTP transitions to **RTP_STATE_DELETE**, this bit is unset.

SYSTEM CALL BUFFER VALIDATION

By default any user buffer passed to a system call will be validated to ensure that it belongs to the RTP's memory space. This validation is a lengthy operation which adds to the system call overhead. The buffer validation can be turned off for a specific RTP by spawning it with the option **RTP_BUFFER_VAL_OFF** (0x20) set. However this leaves a potential security hole so this option should be used only once the application code is properly debugged.

SMP CONSIDERATIONS

By default RTP tasks inherit the CPU affinity setting of the task that created the RTP. If the parent task has no specific CPU affinity (i.e. it can execute on any available CPU and may migrate from one CPU to the other during its lifetime) then the RTP's tasks have no specific CPU affinity either. If the parent task has its affinity set to a given CPU then, by default, the RTP tasks inherit this affinity and execute only on the same CPU as the RTP's parent task.

By using the **rtpSpawn()**'s option **RTP_CPU_AFFINITY_NONE** it is possible to create a RTP which tasks have no specific CPU affinity even though the RTP's parent task may have a specific CPU affinity.

INCLUDE FILES	rtpLib.h
SEE ALSO	rtpShow , rtpUtilLib , rtpSigLib , rtpHookLib , edrLib , sdLib , shlLib

rtpShow

NAME	rtpShow – Real Time Process show routine
ROUTINES	rtpShow() – display information for real time proceses rtpMemShow() – display memory context information for real time proceses rtpHookShow() – display all installed RTP hooks
DESCRIPTION	<p>This library provides routines to display information about the Real Time Processes (RTP) in the system.</p> <p>There are two levels of information that can be obtained: summary and full. Additionally, the request can be applied to a specific RTP, or to all the RTPs within the system. For more information see the rtpShow() manual entry.</p> <p>The information provided by the show routines should be considered an instantaneous snapshot of the system. This function is only designed as a diagnostic aid. Programmatic access to RTP information is provided through the rtpUtilLib functions (for example, rtpInfoGet()). Refer to the rtpUtilLib manual entry for these functions.</p> <p>The rtpShow() routine may be called only from the C interpreter shell. To display information from the command interpreter shell, use rtp or ps.</p>
CONFIGURATION	To use the RTP show routine library, configure VxWorks with the INCLUDE_RTP_SHOW component.
INCLUDE FILES	rtpLib.h
SEE ALSO	rtpLib , rtpUtilLib , the VxWorks programmer guides.

rtpSigLib

NAME	rtpSigLib – RTP software signal facility library
ROUTINES	rtpTaskKill() – send a signal to a task rtpTaskSigqueue() – send a queued signal to a task rtpKill() – send a signal to a RTP rtpSigqueue() – send a queued signal to a RTP
DESCRIPTION	<p>This library provides the signal interfaces for Real Time Processes (RTPs) and tasks within RTPs. Signals alter the execution flow of tasks by communicating asynchronous events within or between task contexts. Any task or interrupt service can "raise" (or send) a signal to a particular task. The task being signaled will immediately suspend its current thread of execution and invoke a task-specified "signal handler" routine. The signal handler can be a user-supplied routine that is bound to a specific signal and performs whatever actions are necessary whenever the signal is received.</p> <p>Signals are most appropriate for error and exception handling, rather than as a general purpose inter-task communication mechanism.</p>
INCLUDE FILES	rtpLib.h
SEE ALSO	rtpLib , POSIX 1003.1b documentation

rtpUtilLib

NAME	rtpUtilLib – Real Time Process Utility library
ROUTINES	rtpInfoGet() – Get specific information on an RTP rtpSymTblIdGet() – Get the symbol table ID of an RTP
DESCRIPTION	<p>This library provides utilities to support the Real Time Process (RTP) feature. The utilities provide ways for applications to access information regarding the RTP.</p>
CONFIGURATION	<p>The selection of the INCLUDE_RTP feature will include these utilities into the VxWorks image.</p>
INCLUDE FILES	rtpLib.h
SEE ALSO	rtpLib , rtpSpawn()

salClient

NAME	salClient – socket application client library
ROUTINES	salOpen() – establish communication with a named socket-based server salSocketFind() – find sockets for a named socket-based server salNameFind() – find services with the specified name salCall() – invoke a socket-based server
DESCRIPTION	This portion of the Socket Application Library (SAL) provides the infrastructure for implementing a socket-based client application. The routines provided by SAL allow client applications to communicate easily with socket-based server applications that are registered with the Socket Name Service (SNS). Some routines can also be used to communicate with unregistered server applications. SAL routines assume connection oriented message based communications. Although it could provide support for all protocols with the above features, the current implementation is supporting only local (single node) inter process communication using the COMP (Connection Oriented Message Passing) protocol and distributed (multi-node) inter process communication using the TIPC (Transparent Inter-Process Communication) protocol.

SAL Client

The SAL client API allows a client application to communicate with a specified server application by using socket descriptors. A client application can utilize SAL routines to communicate with different server applications in succession, or create multiple SAL clients that are each linked to a different server.

A client application typically calls **salOpen()** to configure a socket descriptor associated with a named server application. **salOpen()** simplifies the procedures needed to initialize the socket and its connection to the server. The server can be easily identified by a name, represented by a character string. The client application can then communicate with the server by passing the socket descriptor to standard socket API routines, such as **send()** and **recv()**. Alternatively, the client application can perform a **send()** and **recv()** as a single operation using **salCall()**. When the client application no longer needs to communicate with a server it calls **close()** to close the socket to the server.

A client application can utilize **salSocketFind()** to exercise more control over the establishment of communication with a server, as an alternative to using **salOpen()**. **salSocketFind()** can be used to determine the socket addresses related to a server, and then create a socket to communicate with the server. The client can therefore choose the server socket address or addresses that better suits its needs. A client can also use **salNameFind()** to identify one or more services based on a search pattern. Therefore, the client does not need to know the exact name of a service and, in case multiple names are found, it can choose which ones to use.

Because normal socket descriptors are used, the client application also has access to all of the standard socket API.

EXAMPLE

The following code illustrates how to create a client that utilizes an "ping" service which simply returns each incoming message to the sender. The maximum size of a message is limited to **MAX_PING_SIZE** bytes. This service uses the connection-based COMP socket protocol.

```

/* This routine creates and runs a client of the ping service. */

#include "vxWorks.h"
#include "dsi/salClient.h"

#define MAX_PING_SIZE 72

STATUS pingClient
(
    char * message,           /* message buffer */
    int  msgSize              /* size of message */
)
{
    char reply[MAX_PING_SIZE]; /* reply buffer */
    int replySize;             /* size of reply */
    int sockfd;                /* socket file descriptor */

    /* set up client connection to PING server */

    if ((sockfd = salOpen ("ping")) < 0)
    {
        return ERROR;
    }

    /* send message to PING server and get reply */

    replySize = salCall (sockfd, message, msgSize,
                        reply, sizeof (reply));

    /* tear down client connection to PING server */

    if (close (sockfd) <0)
        return ERROR;

    /* check that reply matches message */

    if ((replySize != msgSize) || (memcmp (message, reply, msgSize) !=
0))
    {
        return ERROR;
    }

    return OK;
}

```

CONFIGURATION To use the SAL client library, configure VxWorks with the **INCLUDE_SAL_CLIENT** component.

INCLUDE FILES **salClient.h**

SEE ALSO **salServer**, **snsLib**

salServer

NAME	salServer – socket application server library
ROUTINES	salCreate() – create a named socket-based server salDelete() – delete a named socket-based server salServerRtnSet() – configures the processing routine with the SAL server salRun() – activate a socket-based server salRemove() – Remove service from SNS by name
DESCRIPTION	This portion of the Socket Application Library (SAL) provides the infrastructure for implementing a socket-based server application. The data structures and routines provided by SAL allow the application to communicate easily with socket-based client applications that locate the server using the Socket Name Service (SNS).

SAL Server ID

The "SAL Server ID" refers to an internal data structure that is used by many routines in the SAL server library. The server data structure allows a server application to provide service to any number of client applications. A server application normally utilizes a single SAL server in its main task, but it is free to spawn additional tasks to handle the processing for individual clients if parallel processing of client requests is required.

Main Capabilities

A server application typically calls **salCreate()** to configure a SAL server with one or more sockets that are then registered with SNS under a specified service identifier. The number of sockets created depends on which address families, socket types, and socket protocols are specified by the server application. The current implementation supports only connection-oriented message based socket types. Although it could provide support for all protocols with the above features, the current implementation is supporting both local (single node) inter process communication using the COMP (Connection Oriented Message passing) protocol and distributed (multi-node) inter process communication using the TIPC (Transparent Inter-Process Communication) protocol. The socket addresses used for the server's sockets are selected automatically and cannot be specified by the server application using **salCreate()**.

Once created, a SAL server must be configured with one or more processing routines before it is activated.

- The "accept" routine is invoked whenever an active socket is created as the result of a new client connecting to the server.

- The "read" routine is invoked whenever an active socket is ready for reading or can no longer be read.

Configuring of the processing routines is accomplished by calling the **salServerRtnSet()** function.

If no routine is supplied, the service will not be activated.

Activation of a SAL server is accomplished by calling **salRun()**. A SAL server runs indefinitely once it has been activated, monitoring the activities on its connections and calling the appropriate processing routines as needed. The SAL server becomes deactivated only at the request of the server application (through the processing routines) or if an unexpected error is detected by **salRun()**.

Once a SAL server has been deactivated the server application calls **salDelete()** to close the server's sockets and deregister the service identifier from SNS.

Processing Routines

The "accept" routine is utilized by any server application that incorporates passive (i.e. listening) sockets into the SAL server. The routine should determine if the connection should be accepted and the new socket added to the SAL server. The routine can return the following values:

SAL_SOCKET_KEEP

the SAL server has accepted the new connection and the new socket should be added to the SAL server.

SAL_SOCKET_CLOSE

the routine is requesting the SAL server to close the socket.

SAL_SOCKET_IGNORE

the SAL server will not add the new socket but it will not close it. This could be because the user application is going to have the socket managed by another task or because it has already closed the socket.

Any other value is considered as an error and deactivates the SAL server.

If a SAL server is not configured with an accept routine **salRun()** uses a default routine that automatically approves of the socket and adds it to the server.

The "read" routine is utilized by any server application that incorporates active sockets into the SAL server. The routine should read the specified socket and process the input accordingly, possibly generating a response. The read routine should return an appropriate value to let **salRun()** know what to do with the socket or to the SAL server.

SAL_SOCKET_CLOSE

the SAL server closes the socket and removes it the from server.

SAL_SOCKET_IGNORE

the SAL server removes the socket from the list without closing it. This might be useful when the application requires another task to take care of the socket.

SAL_SOCKET_KEEP

the socket is kept in the SAL server.

SAL_RUN_TERMINATE

salRun() is terminated, with an **OK** return value. The sockets are not closed.

Any other value is considered as an error and deactivates the SAL server.

The read routine should close the socket and return **SAL_SOCKET_IGNORE**, or ask the SAL server to close the socket (by returning **SAL_SOCKET_CLOSE**), if it detects that the socket connection has been closed by the client. This state is normally indicated by a read operation that receives zero bytes.

If a SAL server is not configured with a read routine and active sockets are present, **salRun()** uses a default routine that deactivates the server with an error.

NOTE

Care must be taken to ensure that a processing routine does not cause **salRun()** to block, otherwise the actions of a single client can halt the server's main task and thereby deny use of the server to other clients. One solution is to use the **MSG_DONTWAIT** flag when reading or writing an active socket; an alternative solution is to use a distinct task for each active socket and not incorporate them into the SAL server.

EXAMPLE

The following code illustrates how to create a server that implements a "ping" service which simply returns each incoming message to the sender. The service satisfies the first **MAX_REQ_COUNT** requests only. Once it has reached the threshold it terminates. The maximum size of a message is limited to **MAX_PING_SIZE** bytes. This service uses the connection-based COMP socket protocol.

```
#include "vxWorks.h"
#include "sockLib.h"
#include "dsi/salServer.h"

/* defines */

#define MAX_PING_SIZE 72      /* max message size */
#define MAX_REQ_COUNT 5      /* max number of client requests */

/* forward declarations */

LOCAL SAL_RTN_STATUS pingServerRead (int sockfd, void * pData);

/* This routine creates and runs the server for the ping service. */

STATUS pingServer (void)
{
    SAL_SERVER_ID serverId;          /* server structure */
    STATUS result;                   /* return value */
    int count;                       /* counter */

    /* create server socket & register service with SNS */

    if ((serverId = salCreate ("ping", AF_LOCAL, SOCK_SEQPACKET, 0,
                              NULL, 0)) == NULL)
```



```

    {
        return ERROR;
    }

/* configure read routine for server */
salServerRtnSet (serverId, SAL_RTN_READ, pingServerRead);

/* request counter initialized */

count = 0;

/* activate the server (never returns unless a fatal error occurs */
/* or the application processing routine requests a termination) */

result = salRun (serverId, &count);

/* close server socket & deregister service from SNS */

salDelete (serverId);

return result;
}

/* This is the read routine for the ping server. */
LOCAL SAL_RTN_STATUS pingServerRead
(
    int sockfd,                                /* active socket to read */
    void * pData                               /* user data */
)
{
    char message[MAX_PING_SIZE];              /* buffer for message */
    int msgSize;                               /* size of message */
    int * pCounter;                            /* request counter */

/* get message from specified client */

msgSize = recv (sockfd, message, sizeof (message), MSG_DONTWAIT);

if (msgSize <= 0)
{
    /* client connection has been closed by client or has failed */

    return SAL SOCK_CLOSE;
}

/* send message back to client */

if (send (sockfd, message, msgSize, MSG_DONTWAIT) < 0)
{
    /* client connection has failed */

    close (sockfd);
    return SAL SOCK_IGNORE;
}

```

```
    }

    pCounter = pData;

    if (*pCounter++ >= MAX_REQ_COUNT)
        return SAL_RUN_TERMINATE;

    /* indicate that client connection is still OK */

    return SAL_SOCKET_KEEP;
}
```

CONFIGURATION	To use the SAL server library, configure VxWorks with the INCLUDE_SAL_SERVER component.
INCLUDE FILES	salServer.h
SEE ALSO	salClient , snsLib

sbeVxbEnd

NAME	sbeVxbEnd – Broadcom/Sibyte BCM1250 VxBus END driver
ROUTINES	sbeRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver from the on-board ethernet in the Broadcom SB1 series BCM1250 and BCM1480 boards. The SB1 ethernet provides dual RX and TX channels (currently this driver uses only one RX and TX channel each), RX TCP/IP checksum offload, RMON statistics counting, a 512-bit multicast hash table, and an MDIO interface to a GMII compliant PHY.</p> <p>Early revisions of the SBE controller have a major limitation concerning DMA alignment requirements: when performing scatter/gather DMA, all but the first buffer must be cache-line aligned, and intermediate buffers must all be exactly a multiple of the cache line size in length. These requirements are fairly stringent, and when sourcing TCP or UDP traffic, the current VxWorks TCP/IP stack almost never generates multi-fragment transmissions that satisfy these requirements. This results in a potential performance penalty in many cases, except in packet forwarding scenarios (when forwarding IP packets, for example, packets are almost always contained in single buffers that match the SB1's alignment requirements).</p> <p>In BCM1250 devices with PERIPH_REV3 or later (e.g. BCM1250 stepping C0) and the 1255/1280/1455/1480 devices, a new descriptor format has been added that allows unaligned DMA operations that are more compatible with the VxWorks stack. This format will be used if a device which supports it is detected.</p>

BOARD LAYOUT The Broadcom SB1 interfaces are integrated into the BCM1250/BCM1480 processors.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **sbeRegister()** function, which registers the driver with VxBus.

The SBE controller also supports jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of a buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface sbe0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "sbe", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

The SBE controller also supports interrupt coalescing. This driver has coalescing support, which is disabled by default so that the **out of the box** configuration has the smallest interrupt latency. Coalescing can be enabled on a per-interface basis using parameter overrides in the **hwconf.c** file, in the same way as jumbo frame support. In addition to turning the coalescing support on and off, the timeout and packet count values can be set:

```
{ "sbe", 0, "coalesceEnable", VXB_PARAM_INT32, {(void *)1} }  
{ "sbe", 0, "coalesceRxTicks", VXB_PARAM_INT32, {(void *)200} }  
{ "sbe", 0, "coalesceRxPkts", VXB_PARAM_INT32, {(void *)16} }  
{ "sbe", 0, "coalesceTxTicks", VXB_PARAM_INT32, {(void *)800} }  
{ "sbe", 0, "coalesceTxPkts", VXB_PARAM_INT32, {(void *)32} }
```

If only the *coalesceEnable* property is set, the driver will use default timeout and packet count values as shown above. Specifying alternate values via the BSP will override the defaults.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one non-VxBus external support function:

```
void sysSbeEnetAddrGet (int unit, char * pAddr);
```

This routine is needed to obtain the station address from the BSP. Eventually, this function will be deprecated once VxBus-centric support for obtaining station addresses from on-board NVRAM is available.

INCLUDE FILES **sbeVxbEnd.h sb1Lib.h end.h endLib.h netBufLib.h muxLib.h**

SEE ALSO vxBus, **ifLib**, "Writing an Enhanced Network Driver", "Broadcom BCM1250 User's Manual"

scMemVal

NAME	scMemVal – helper routines to validate system call parameters
ROUTINES	scMemValEnable() – enable or disable pointer/buffer validation in system calls scMemValidate() – validate an address range passed to a system call routine
DESCRIPTION	This library provides for a buffer validation routine, scMemValidate() . Parameters passed to a system call routine need to be validated. As part of this validation process it is necessary to guarantee that a pointer passed to the system call routine points to memory belonging to the calling RTP, and that the kernel system call code can access this memory, whether it needs to read from it, or write to it. This routine is to be used only to validate pointers passed as parameters to system calls. For information on how to add custom APIs to the system call interface, refer to VxWorks Programmers' Guide, "Kernel", "Adding Custom APIs to the System Call Interface".
NOTE	The routine scMemValidate() is to be called only within code included when RTP support is included. Failure to do so will drag the whole RTP support libraries into the kernel even if the component INCLUDE_RTP is not defined.
INCLUDE FILES	scMemVal.h
SEE ALSO	rtpSpawn() , the VxWorks programmer guides.

schedPxLib

NAME	schedPxLib – scheduling library (POSIX)
ROUTINES	sched_setparam() – set a task's priority (POSIX) sched_getparam() – get the scheduling parameters for a specified task (POSIX) sched_setscheduler() – set scheduling policy and scheduling parameters (POSIX) sched_getscheduler() – get the current scheduling policy (POSIX) sched_yield() – relinquish the CPU (POSIX) sched_get_priority_max() – get the maximum priority (POSIX) sched_get_priority_min() – get the minimum priority (POSIX) sched_rr_get_interval() – get the current time slice (POSIX)
DESCRIPTION	This library provides POSIX-compliance scheduling routines. The routines in this library allow the user to get and set priorities and scheduling schemes, get maximum and minimum priority values, and get the time slice if round-robin scheduling is enabled.

The POSIX standard specifies a priority numbering scheme in which higher priorities are indicated by larger numbers. The VxWorks native numbering scheme is the reverse of this, with higher priorities indicated by smaller numbers. For example, in the VxWorks native priority numbering scheme, the highest priority task has a priority of 0.

In VxWorks, POSIX scheduling interfaces are implemented using the POSIX priority numbering scheme. This means that the priority numbers used by this library *do not* match those reported and used in all the other VxWorks components. It is possible to change the priority numbering scheme used by this library by setting the global variable **posixPriorityNumbering**. If this variable is set to **FALSE**, the VxWorks native numbering scheme (small number = high priority) is used, and priority numbers used by this library will match those used by the other portions of VxWorks.

The routines in this library are compliant with POSIX 1003.1b. In particular, task priorities are set and reported through the structure **sched_setparam**, which has a single member:

```
struct sched_param          /* Scheduling parameter structure */
{
    int sched_priority;      /* scheduling priority */
};
```

POSIX 1003.1b specifies this indirection to permit future extensions through the same calling interface. For example, because **sched_setparam()** takes this structure as an argument (rather than using the priority value directly) its type signature need not change if future schedulers require other parameters.

INCLUDE FILES **sched.h**

SEE ALSO POSIX 1003.1b document, **taskLib**

scsi1Lib

NAME **scsi1Lib** – Small Computer System Interface (SCSI) library (SCSI-1)

ROUTINES

DESCRIPTION This library implements the Small Computer System Interface (SCSI) protocol in a controller-independent manner. It implements only the SCSI initiator function; the library does not support a VxWorks target acting as a SCSI target. Furthermore, in the current implementation, a VxWorks target is assumed to be the only initiator on the SCSI bus, although there may be multiple targets (SCSI peripherals) on the bus.

The implementation is transaction based. A transaction is defined as the selection of a SCSI device by the initiator, the issuance of a SCSI command, and the sequence of data, status, and message phases necessary to perform the command. A transaction normally completes

with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. If the status from the target is "Check Condition," the transaction continues; the initiator issues a "Request Sense" command to gain more information on the exception condition reported.

Many of the subroutines in **scsi1Lib** facilitate the transaction of frequently used SCSI commands. Individual command fields are passed as arguments from which SCSI Command Descriptor Blocks are constructed, and fields of a **SCSI_TRANSACTION** structure are filled in appropriately. This structure, along with the **SCSI_PHYS_DEV** structure associated with the target SCSI device, is passed to the routine whose address is indicated by the **scsiTransact** field of the **SCSI_CTRL** structure associated with the relevant SCSI controller.

The function variable **scsiTransact** is set by the individual SCSI controller driver. For off-board SCSI controllers, this routine rearranges the fields of the **SCSI_TRANSACTION** structure into the appropriate structure for the specified hardware, which then carries out the transaction through firmware control. Drivers for an on-board SCSI-controller chip can use the **scsiTransact()** routine in **scsiLib** (which invokes the **scsi1Transact()** routine in **scsi1Lib**), as long as they provide the other functions specified in the **SCSI_CTRL** structure.

Note that no disconnect/reconnect capability is currently supported.

SUPPORTED SCSI DEVICES

The **scsi1Lib** library supports use of SCSI peripherals conforming to the standards specified in *"Common Command Set (CCS) of the SCSI, Rev. 4.B."* Most SCSI peripherals currently offered support CCS. While an attempt has been made to have **scsi1Lib** support non-CCS peripherals, not all commands or features of this library are guaranteed to work with them. For example, auto-configuration may be impossible with non-CCS devices, if they do not support the INQUIRY command.

Not all classes of SCSI devices are supported. However, the **scsiLib** library provides the capability to transact any SCSI command on any SCSI device through the FIOSCSICOMMAND function of the **scsiIoctl()** routine.

Only direct-access devices (disks) are supported by a file system. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

CONFIGURING VXWORKS

To use the SCSI-1 library, configure VxWorks with the **INCLUDE_SCSI1** component.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);  /* parameters are controller specific */
```

The conceptual difference between the two routines is that **xxCtrlCreate()** alloc's memory for the **xx SCSI_CTRL** data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the **xx SCSI_CTRL** structure are initialized by **xxCtrlInit()** and any necessary registers are written on the SCSI controller to effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with **scsiPhysDevCreate()** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
(
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller info */
    int devBusId,          /* device's SCSI bus ID */
    int devLUN,            /* device's logical unit number */
    int reqSenseLength,    /* length of REQUEST SENSE data dev returns */
    int devType,           /* type of SCSI device */
    BOOL removable,        /* whether medium is removable */
    int numBlocks,         /* number of blocks on device */
    int blockSize          /* size of a block in bytes */
)
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength

If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType

If -1, issue an **INQUIRY** to determine the device type.

numBlocks, blockSize

If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and a specified number of blocks. The **scsiBlkDevCreate()** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
(
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device info */
    int numBlocks, /* number of blocks in block device */
    int blockOffset /* address of first block in volume */
)
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (such as dosFs) has been initialized on the partition. For more information, see the documentation in **dosFsLib**.

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi1Lib** library provides routines that implement many common SCSI commands. Still, there are situations that require commands that are not supported by **scsi1Lib** (for example, writing software to control non-direct access devices). Arbitrary commands are handled with the FIOSCSICOMMAND option to **scsiIoctl()**. The *arg* parameter for FIOSCSICOMMAND is a pointer to a valid SCSI_TRANSACTION structure. Typically, a call to **scsiIoctl()** is written as a subroutine of the form:

```
STATUS myScsiCommand
(
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    char *          buffer,        /* ptr to data buffer */
    int             bufLength,     /* length of buffer in bytes */
    int             someParam      /* param. specifiable in cmd block */
)
{
    SCSI_COMMAND myScsiCmdBlock; /* SCSI command byte array */
    SCSI_TRANSACTION myScsiXaction; /* info on a SCSI transaction */

    /* fill in fields of SCSI_COMMAND structure */

    myScsiCmdBlock[0] = MY_COMMAND_OPCODE; /* the required opcode */
    .
    myScsiCmdBlock[X] = (UINT8) someParam; /* for example */
    .
    myScsiCmdBlock[N-1] = MY_CONTROL_BYTE; /* typically == 0 */

    /* fill in fields of SCSI_TRANSACTION structure */

    myScsiXaction.cmdAddress = myScsiCmdBlock;
    myScsiXaction.cmdLength = <# of valid bytes in myScsiCmdBlock>;
    myScsiXaction.dataAddress = (UINT8 *) buffer;
    myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
    myScsiXaction.dataLength = bufLength;
    myScsiXaction.cmdTimeout = timeout in usec;

    /* if dataDirection is O_RDONLY, and the length of the input data is
     * variable, the following parameter specifies the byte # (min == 0)
     * of the input data which will specify the additional number of
     * bytes available
     */

    myScsiXaction.addLengthByte = X;

    if (scsiIoctl(pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
        return (OK);
}
```



```
else
    /* optionally perform retry or other action based on value of
     * myScsiXaction.statusByte
     */
    return (ERROR);
}
```

INCLUDE FILES **scsiLib.h, scsi1Lib.h**

SEE ALSO **dosFsLib**, *American National Standards for Information Systems - Small Computer, System Interface (SCSI)*, ANSI X3.131-1986, the VxWorks programmer guides.

scsi2Lib

NAME **scsi2Lib** – Small Computer System Interface (SCSI) library (SCSI-2)

ROUTINES **scsiIfInit()** – initialize the SCSI-2 interface to **scsiLib**
scsiTargetOptionsSet() – set options for one or all SCSI targets
scsiTargetOptionsGet() – get options for one or all SCSI targets
scsiTargetOptionsShow() – display options for specified SCSI target
scsiPhysDevShow() – show status information for a physical device
scsiCacheSynchronize() – synchronize the caches for data coherency
scsiIdentMsgBuild() – build an identification message
scsiIdentMsgParse() – parse an identification message
scsiMsgOutComplete() – perform post-processing after a SCSI message is sent
scsiMsgOutReject() – perform post-processing when an outgoing message is rejected
scsiMsgInComplete() – handle a complete SCSI message received from the target
scsiSyncXferNegotiate() – initiate or continue negotiating transfer parameters
scsiWideXferNegotiate() – initiate or continue negotiating wide parameters
scsiThreadInit() – perform generic SCSI thread initialization
scsiCacheSnoopEnable() – inform SCSI that hardware snooping of caches is enabled
scsiCacheSnoopDisable() – inform SCSI that hardware snooping of caches is disabled

DESCRIPTION This library implements the Small Computer System Interface (SCSI) protocol in a controller-independent manner. It implements only the SCSI initiator function as defined in the SCSI-2 ANSI specification. This library does not support a VxWorks target acting as a SCSI target.

The implementation is transaction based. A transaction is defined as the selection of a SCSI device by the initiator, the issuance of a SCSI command, and the sequence of data, status, and message phases necessary to perform the command. A transaction normally completes with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. If the status from the target is "Check Condition," the transaction continues; the

initiator issues a "Request Sense" command to gain more information on the exception condition reported.

Many of the subroutines in **scsi2Lib** facilitate the transaction of frequently used SCSI commands. Individual command fields are passed as arguments from which SCSI Command Descriptor Blocks are constructed, and fields of a **SCSI_TRANSACTION** structure are filled in appropriately. This structure, along with the **SCSI_PHYS_DEV** structure associated with the target SCSI device, is passed to the routine whose address is indicated by the **scsiTransact** field of the **SCSI_CTRL** structure associated with the relevant SCSI controller. The above mentioned structures are defined in **scsi2Lib.h**.

The function variable **scsiTransact** is set by the individual SCSI controller driver. For off-board SCSI controllers, this routine rearranges the fields of the **SCSI_TRANSACTION** structure into the appropriate structure for the specified hardware, which then carries out the transaction through firmware control. Drivers for an on-board SCSI-controller chip can use the **scsiTransact()** routine in **scsiLib** (which invokes the **scsi2Transact()** routine in **scsi2Lib**), as long as they provide the other functions specified in the **SCSI_CTRL** structure.

SCSI TRANSACTION TIMEOUT

Associated with each transaction is a time limit (specified in microseconds, but measured with the resolution of the system clock). If the transaction has not completed within this time limit, the SCSI library aborts it; the called routine fails with a corresponding error code. The timeout period includes time spent waiting for the target device to become free to accept the command.

The semantics of the timeout should guarantee that the caller waits no longer than the transaction timeout period, but in practice this may depend on the state of the SCSI bus and the connected target device when the timeout occurs. If the target behaves correctly according to the SCSI specification, proper timeout behavior results. However, in certain unusual cases—for example, when the target does not respond to an asserted ATN signal—the caller may remain blocked for longer than the timeout period.

If the transaction timeout causes problems in your system, you can set the value of either or both the global variables "scsi{Min,Max}Timeout". These specify (in microseconds) the global minimum and maximum timeout periods, which override (clip) the value specified for a transaction. They may be changed at any time and affect all transactions issued after the new values are set. The range of both these variable is 0 to 0xffffffff (zero to about 4295 seconds).

SCSI TRANSACTION PRIORITY

Each transaction also has an associated priority used by the SCSI library when selecting the next command to issue when the SCSI system is idle. It chooses the highest priority transaction that can be dispatched on an available physical device. If there are several equal-priority transactions available, the SCSI library uses a simple round-robin scheme to avoid favoring the same physical device.

Priorities range from 0 (highest) to 255 (lowest), which is the same as task priorities. The priority `SCSI_THREAD_TASK_PRIORITY` can be used to give the transaction the same priority as the calling task (this is the method used internally by this SCSI-2 library).

SUPPORTED SCSI DEVICES

This library requires peripherals that conform to the SCSI-2 ANSI standard; in particular, the INQUIRY, REQUEST SENSE, and TEST UNIT READY commands must be supported as specified by this standard. In general, the SCSI library is self-configuring to work with any device that meets these requirements.

Peripherals that support identification and the SCSI message protocol are strongly recommended as these provide maximum performance.

In theory, all classes of SCSI devices are supported. The **scsiLib** library provides the capability to transact any SCSI command on any SCSI device through the `FIOSCSICOMMAND` function of the `scsiIoctl()` routine (which invokes the `scsi2Ioctl()` routine in **scsi2Lib**).

Only direct-access devices (disks) are supported by file systems like `dosFs`, and `rawFs`. These file systems employ routines in **scsiDirectLib** (most of which are described in **scsiLib** but defined in **scsiDirectLib**). In the case of sequential-access devices (tapes), higher-level tape file systems, like `tapeFs`, make use of **scsiSeqLib**. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

DISCONNECT/RECONNECT SUPPORT

The target device can be disconnected from the SCSI bus while it carries out a SCSI command; in this way, commands to multiple SCSI devices can be overlapped to improve overall SCSI throughput. There are no restrictions on the number of pending, disconnected commands or the order in which they are resumed. The SCSI library serializes access to the device according to the capabilities and status of the device (see the following section).

Use of the disconnect/reconnect mechanism is invisible to users of the SCSI library. It can be enabled and disabled separately for each target device (see `scsiTargetOptionsSet()`). Note that support for disconnect/reconnect depends on the capabilities of the controller and its driver (see below).

TAGGED COMMAND QUEUEING SUPPORT

If the target device conforms to the ANSI SCSI-2 standard and indicates (using the INQUIRY command) that it supports command queuing, the SCSI library allows new commands to be started on the device whenever the SCSI bus is idle. That is, it executes multiple commands concurrently on the target device. By default, commands are tagged with a SIMPLE QUEUE TAG message. Up to 256 commands can be executing concurrently.

The SCSI library correctly handles contingent allegiance conditions that arise while a device is executing tagged commands. (A contingent allegiance condition exists when a target device is maintaining sense data that the initiator should use to correctly recover from an

error condition.) It issues an untagged REQUEST SENSE command, and stops issuing tagged commands until the sense recovery command has completed.

For devices that do not support command queuing, the SCSI library only issues a new command when the previous one has completed. These devices can only execute a single command at once.

Use of tagged command queuing is normally invisible to users of the SCSI library. If necessary, the default tag type and maximum number of tags may be changed on a per-target basis, using **scsiTargetOptionsSet()**.

SYNCHRONOUS TRANSFER PROTOCOL SUPPORT

If the SCSI controller hardware supports the synchronous transfer protocol, **scsiLib** negotiates with the target device to determine whether to use synchronous or asynchronous transfers. Either VxWorks or the target device may start a round of negotiation. Depending on the controller hardware, synchronous transfer rates up to the maximum allowed by the SCSI-2 standard (10 Mtransfers/second) can be used.

Again, this is normally invisible to users of the SCSI library, but synchronous transfer parameters may be set or disabled on a per-target basis by using **scsiTargetOptionsSet()**.

WIDE DATA TRANSFER SUPPORT

If the SCSI controller supports the wide data transfer protocol, **scsiLib** negotiates wide data transfer parameters with the target device, if that device also supports wide transfers. Either VxWorks or the target device may start a round of negotiation. Wide data transfer parameters are negotiated prior to the synchronous data transfer parameters, as specified by the SCSI-2 ANSI specification. In conjunction with synchronous transfer, up to a maximum of 20MB/sec. can be attained.

Wide data transfer negotiation is invisible to users of this library, but it is possible to enable or disable wide data transfers and the parameters on a per-target basis by using **scsiTargetOptionsSet()**.

SCSI BUS RESET

The SCSI library implements the ANSI "hard reset" option. Any transactions in progress when a SCSI bus reset is detected fail with an error code indicating termination due to bus reset. Any transactions waiting to start executing are then started normally.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);  /* parameters are controller specific */
```

The conceptual difference between the two routines is that **xxCtrlCreate()** alloc's memory for the **xx SCSI_CTRL** data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the **xx SCSI_CTRL** structure are initialized by **xxCtrlInit()** and any necessary registers are written on the SCSI controller to

effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING VXWORKS

To use the SCSI-2 library, configure VxWorks with the **INCLUDE_SCSI2** component.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with **scsiPhysDevCreate()** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
(
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller info */
    int devBusId,          /* device's SCSI bus ID */
    int devLUN,            /* device's logical unit number */
    int reqSenseLength,    /* length of REQUEST SENSE data dev returns */
    int devType,            /* type of SCSI device */
    BOOL removable,        /* whether medium is removable */
    int numBlocks,          /* number of blocks on device */
    int blockSize          /* size of a block in bytes */
)
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength

If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType

This parameter is ignored: an **INQUIRY** command is used to ascertain the device type. A value of **NONE** (-1) is the recommended placeholder.

numBlocks, blockSize

If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON DIRECT-ACCESS BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and specified number of blocks. The **scsiBlkDevCreate()** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
(
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device info */
    int numBlocks, /* number of blocks in block device */
    int blockOffset /* address of first block in volume */
)
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING DISK FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (for example, dosFs, or rawFs) has been initialized on the partition. For more information, see the relevant documentation in **dosFsLib** or **rawFsLib**.

USING A SEQUENTIAL-ACCESS BLOCK DEVICE

The entire volume (tape) on a sequential-access block device is treated as a single raw file. This raw file is made available to higher-level layers like tapeFs by the **scsiSeqDevCreate()** routine, described in **scsiSeqLib**. The **scsiSeqDevCreate()** routine is called once for a given SCSI physical device.

```
SEQ_DEV *scsiSeqDevCreate
(
    SCSI_PHYS_DEV *pScsiPhysDev /* ptr to SCSI physical device info */
)
```

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, and **scsiSeqLib** libraries collectively provide routines that implement all mandatory SCSI-2 direct-access and sequential-access commands. Still, there are situations that require commands that are not supported by these libraries (for example, writing software that needs to use an optional SCSI-2 command). Arbitrary commands are handled with the FIOSCSICOMMAND option to **scsiIoctl()**. The *arg* parameter for FIOSCSICOMMAND is a pointer to a valid SCSI_TRANSACTION structure. Typically, a call to **scsiIoctl()** is written as a subroutine of the form:

```
STATUS myScsiCommand
(
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    char *          buffer,        /* ptr to data buffer */
    int             bufLength,     /* length of buffer in bytes */
    int             someParam      /* param. specifiable in cmd block */
)

{
    SCSI_COMMAND myScsiCmdBlock; /* SCSI command byte array */
    SCSI_TRANSACTION myScsiXaction; /* info on a SCSI transaction */

    /* fill in fields of SCSI_COMMAND structure */

    myScsiCmdBlock[0] = MY_COMMAND_OPCODE; /* the required opcode */
    .
    myScsiCmdBlock[X] = (UINT8) someParam; /* for example */
    .
    myScsiCmdBlock[N-1] = MY_CONTROL_BYTE; /* typically == 0 */

    /* fill in fields of SCSI_TRANSACTION structure */

    myScsiXaction.cmdAddress = myScsiCmdBlock;
    myScsiXaction.cmdLength = <# of valid bytes in myScsiCmdBlock>;
```

```
myScsiXaction.dataAddress    = (UINT8 *) buffer;
myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
myScsiXaction.dataLength    = bufLength;
myScsiXaction.addLengthByte = 0; /* no longer used */
myScsiXaction.cmdTimeout    = <timeout in usec>;
myScsiXaction.tagType       = SCSI_TAG_{DEFAULT, UNTAGGED,
                                     SIMPLE, ORDERED, HEAD_OF_Q};
myScsiXaction.priority      = [ 0 (highest) to 255 (lowest) ];

if (scsiIoctl (pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
return (OK);
else
    /* optionally perform retry or other action based on value of
    * myScsiXaction.statusByte
    */
return (ERROR);
}
```

INCLUDE FILES **scsiLib.h, scsi2Lib.h**

SEE ALSO **dosFsLib, rawFsLib, tapeFsLib, scsiLib, scsiCommonLib, scsiDirectLib, scsiSeqLib, scsiMgrLib, scsiCtrlLib, American National Standard for Information Systems - Small Computer, System Interface (SCSI-2), ANSI X3T9, the VxWorks programmer guides.**

scsiCommonLib

NAME **scsiCommonLib** – SCSI library common commands for all devices (SCSI-2)

ROUTINES

DESCRIPTION This library contains commands common to all SCSI devices. The content of this library is separated from the other SCSI libraries in order to create an additional layer for better support of all SCSI devices.

Commands in this library include:

Command	Op Code
INQUIRY	(0x12)
REQUEST SENSE	(0x03)
TEST UNIT READY	(0x00)

CONFIGURATION To use the SCSI common commands library, configure VxWorks with the **INCLUDE_SCSI2** component.

INCLUDE FILES **scsiLib.h, scsi2Lib.h**

SEE ALSO **dosFsLib, rawFsLib, tapeFsLib, scsi2Lib**

scsiCtrlLib

NAME **scsiCtrlLib** – SCSI thread-level controller library (SCSI-2)

ROUTINES

DESCRIPTION The purpose of the SCSI controller library is to support basic SCSI controller drivers that rely on a higher level of software in order to manage SCSI transactions. More advanced SCSI I/O processors do not require this protocol engine since software support for SCSI transactions is provided at the SCSI I/O processor level.

This library provides all the high-level routines that manage the state of the SCSI threads and guide the SCSI I/O transaction through its various stages:

- selecting a SCSI peripheral device;
- sending the identify message in order to establish the ITL nexus;
- cycling through information transfer, message and data, and status phases;
- handling bus-initiated reselects.

The various stages of the SCSI I/O transaction are reported to the SCSI manager as SCSI events. Event selection and management is handled by routines in this library.

CONFIGURATION The thread-level controller library is automatically included when the **INCLUDE_SCSI2** component is configured.

INCLUDE FILES **scsiLib.h, scsi2Lib.h**

SEE ALSO **scsiLib, scsi2Lib, scsiCommonLib, scsiDirectLib, scsiSeqLib, scsiMgrLib**, *American National Standard for Information Systems - Small Computer, System Interface (SCSI-2)*, *ANSI X3T9*, the VxWorks programmer guides.

scsiDirectLib

NAME **scsiDirectLib** – SCSI library for direct access devices (SCSI-2)

ROUTINES **scsiStartStopUnit()** – issue a **START_STOP_UNIT** command to a SCSI device
scsiReserve() – issue a **RESERVE** command to a SCSI device

scsiRelease() – issue a RELEASE command to a SCSI device

DESCRIPTION This library contains commands common to all direct-access SCSI devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI direct-access devices.

Commands in this library include:

Command	Op Code
FORMAT UNIT	(0x04)
READ (6)	(0x08)
READ (10)	(0x28)
READ CAPACITY	(0x25)
RELEASE	(0x17)
RESERVE	(0x16)
MODE SELECT (6)	(0x15)
MODE SELECT (10)	(0x55)
MODE SENSE (6)	(0x1a)
MODE SENSE (10)	(0x5a)
START STOP UNIT	(0x1b)
WRITE (6)	(0x0a)
WRITE (10)	(0x2a)

CONFIGURATION The SCSI library for direct access devices is automatically included when the **INCLUDE SCSI2** component is configured.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **rawFsLib**, **scsi2Lib**, the VxWorks programmer guides.

scsiLib

NAME **scsiLib** – Small Computer System Interface (SCSI) library

ROUTINES

- scsiPhysDevDelete()** – delete a SCSI physical-device structure
- scsiPhysDevCreate()** – create a SCSI physical device structure
- scsiPhysDevIdGet()** – return a pointer to a **SCSI_PHYS_DEV** structure
- scsiAutoConfig()** – configure all devices connected to a SCSI controller
- scsiShow()** – list the physical devices attached to a SCSI controller
- scsiBlkDevCreate()** – define a logical partition on a SCSI block device
- scsiBlkDevInit()** – initialize fields in a SCSI logical partition
- scsiBlkDevShow()** – show the **BLK_DEV** structures on a specified physical device
- scsiBusReset()** – pulse the reset signal on the SCSI bus

scsiIoctl() – perform a device-specific I/O control function
scsiFormatUnit() – issue a **FORMAT_UNIT** command to a SCSI device
scsiModeSelect() – issue a **MODE_SELECT** command to a SCSI device
scsiModeSense() – issue a **MODE_SENSE** command to a SCSI device
scsiReadCapacity() – issue a **READ_CAPACITY** command to a SCSI device
scsiRdSecs() – read sector(s) from a SCSI block device
scsiWrtSecs() – write sector(s) to a SCSI block device
scsiTestUnitRdy() – issue a **TEST_UNIT_READY** command to a SCSI device
scsiInquiry() – issue an **INQUIRY** command to a SCSI device
scsiReqSense() – issue a **REQUEST_SENSE** command to a SCSI device and read results

DESCRIPTION	<p>The purpose of this library is to switch SCSI function calls (the common SCSI-1 and SCSI-2 calls listed above) to either scsi1Lib or scsi2Lib, depending upon the SCSI configuration in the Board Support Package (BSP). The normal usage is to configure SCSI-2. However, SCSI-1 is configured when device incompatibilities exist. VxWorks can be configured with either SCSI-1 or SCSI-2, but not both SCSI-1 and SCSI-2 simultaneously.</p> <p>For more information about SCSI-1 functionality, refer to scsi1Lib. For more information about SCSI-2, refer to scsi2Lib.</p>
CONFIGURATION	To use the SCSI system interface library, configure VxWorks with the INCLUDE_SCSI component.
INCLUDE FILES	scsiLib.h , scsi1Lib.h , scsi2Lib.h
SEE ALSO	dosFsLib , rawFsLib , scsi1Lib , scsi2Lib , <i>VxWorks Programmer's Guide: I/O System, Local File Systems</i>

scsiMgrLib

NAME	scsiMgrLib – SCSI manager library (SCSI-2)
ROUTINES	<p> scsiMgrEventNotify() – notify the SCSI manager of a SCSI (controller) event scsiMgrBusReset() – handle a controller-bus reset event scsiMgrCtrlEvent() – send an event to the SCSI controller state machine scsiMgrThreadEvent() – send an event to the thread state machine scsiMgrShow() – show status information for the SCSI manager </p>
DESCRIPTION	<p>This SCSI-2 library implements the SCSI manager. The purpose of the SCSI manager is to manage SCSI threads between requesting VxWorks tasks and the SCSI controller. The SCSI manager handles SCSI events and SCSI threads but allocation and de-allocation of SCSI threads is not the manager's responsibility. SCSI thread management includes dispatching</p>

threads and scheduling multiple threads (which are performed by the SCSI manager), plus allocation and de-allocation of threads (which are performed by routines in **scsi2Lib**).

The SCSI manager is spawned as a VxWorks task upon initialization of the SCSI interface within VxWorks. The entry point of the SCSI manager task is **scsiMgr()**. The SCSI manager task is usually spawned during initialization of the SCSI controller driver. The driver's **xxxCtrlCreateScsi2()** routine is typically responsible for such SCSI interface initializations.

Once the SCSI manager has been initialized, it is ready to handle SCSI requests from VxWorks tasks. The SCSI manager has the following responsibilities:

- It processes requests from client tasks.
- It activates a SCSI transaction thread by appending it to the target device's wait queue and allocating a specified time period to execute a transaction.
- It handles timeout events which cause threads to be aborted.
- It receives event notifications from the SCSI driver interrupt service routine (ISR) and processes the event.
- It responds to events generated by the controller hardware, such as disconnection and information transfer requests.
- It replies to clients when their requests have completed or aborted.

One SCSI manager task must be spawned per SCSI controller. Thus, if a particular hardware platform contains more than one SCSI controller then that number of SCSI manager tasks must be spawned by the controller-driver initialization routine.

CONFIGURATION	The SCSI manager library is automatically configured when INCLUDE_SCSI2 is configured in VxWorks.
INCLUDE FILES	scsiLib.h , scsi2Lib.h
SEE ALSO	scsiLib , scsi2Lib , scsiCommonLib , scsiDirectLib , scsiSeqLib , scsiCtrlLib , <i>American National Standard for Information Systems - Small Computer, System Interface (SCSI-2)</i> , <i>ANSI X3T9</i> , the VxWorks programmer guides.

scsiSeqLib

NAME	scsiSeqLib – SCSI sequential access device library (SCSI-2)
ROUTINES	scsiSeqDevCreate() – create a SCSI sequential device scsiErase() – issue an ERASE command to a SCSI device scsiTapeModeSelect() – issue a MODE_SELECT command to a SCSI tape device scsiTapeModeSense() – issue a MODE_SENSE command to a SCSI tape device

scsiSeqReadBlockLimits() – issue a **READ_BLOCK_LIMITS** command to a SCSI device
scsiRdTape() – read bytes or blocks from a SCSI tape device
scsiWrtTape() – write data to a SCSI tape device
scsiRewind() – issue a **REWIND** command to a SCSI device
scsiReserveUnit() – issue a **RESERVE UNIT** command to a SCSI device
scsiReleaseUnit() – issue a **RELEASE UNIT** command to a SCSI device
scsiLoadUnit() – issue a **LOAD/UNLOAD** command to a SCSI device
scsiWrtFileMarks() – write file marks to a SCSI sequential device
scsiSpace() – move the tape on a specified physical SCSI device
scsiSeqStatusCheck() – detect a change in media
scsiSeqIoctl() – perform an I/O control function for sequential access devices

DESCRIPTION This library contains commands common to all sequential-access SCSI devices. Sequential-access SCSI devices are usually SCSI tape devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI sequential devices.

SCSI commands in this library include:

Command	Op Code
ERASE	(0x19)
MODE SELECT (6)	(0x15)
MODE SENSE (6)	(0x1a)
READ (6)	(0x08)
READ BLOCK LIMITS	(0x05)
RELEASE UNIT	(0x17)
RESERVE UNIT	(0x16)
REWIND	(0x01)
SPACE	(0x11)
WRITE (6)	(0x0a)
WRITE FILEMARKS	(0x10)
LOAD/UNLOAD	(0x1b)

The SCSI routines implemented here operate mostly on a **SCSI_SEQ_DEV** structure. This structure acts as an interface between this library and a higher-level layer. The **SEQ_DEV** structure is analogous to the **BLK_DEV** structure for block devices.

The **scsiSeqDevCreate()** routine creates a **SCSI_SEQ_DEV** structure whose first element is a **SEQ_DEV**, operated upon by higher layers. This routine publishes all functions to be invoked by higher layers and maintains some state information (for example, block size) for tracking SCSI-sequential-device information.

CONFIGURATION The SCSI sequential access device library is automatically included when you configure VxWorks with the **INCLUDE_SCSI2** component.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO `tapeFsLib`, `scsi2Lib`, the VxWorks programmer guides.

sdLib

NAME `sdLib` – shared data API layer

ROUTINES

- `sdCreate()` – create a new shared data region
- `sdOpen()` – open a shared data region for use
- `sdDelete()` – delete a shared data region
- `sdMap()` – map a shared data region into an application or the kernel
- `sdUnmap()` – unmap a shared data region from an application or the kernel
- `sdProtect()` – change the protection attributes of a mapped SD
- `sdInfoGet()` – get specific information about a Shared Data Region
- `sdCreateHookAdd()` – add a hook routine to be called at Shared Data creation
- `sdCreateHookDelete()` – delete a Shared Data creation hook routine
- `sdDeleteHookAdd()` – add a hook routine to be called at Shared Data deletion
- `sdDeleteHookDelete()` – delete a Shared Data deletion hook routine
- `sdGenericHookAdd()` – add a hook routine to be called before Shared Data routine
- `sdGenericHookDelete()` – delete a Shared Data generic hook routine

DESCRIPTION This library provides shared data region management for VxWorks. The purpose of shared data regions is to allow physical memory, or other physical resources such as blocks of memory mapped I/O space to be shared between multiple applications.

To configure shared data management into the system, the component `INCLUDE_SHARED_DATA` must be included in the kernel.

To include display routines for shared data regions the component `INCLUDE_SHOW_ROUTINES` must be configured in conjunction with `INCLUDE_SHARED_DATA`.

CREATION A shared data region can be created via one of two routines:

```
sdOpen (char * name, int options, int mode, UINT32 size,
        off_t64 physAddress, MMU_ATTR attr, void ** pVirtAddress);
```

```
sdCreate (char * name, int options, UINT32 size, off_t64 physAddress,
          MMU_ATTR attr, void ** pVirtAddress);
```

The behavior of `sdOpen` is determined by the value of its *mode* parameter. If the default value of 0 is passed, then a shared data region will not be created.

To create a shared data region using `sdOpen()` the `OM_CREATE` flag must be passed in the *mode* parameter. If just this flag is passed in *mode* and a shared data region with the *name* specified does not already exist in the system the region will be created. However, if a

shared data region *name* already exists, then **sdOpen()** will map that region into the memory context of the calling task and return its **SD_ID**.

If both the **OM_CREATE** and **OM_EXCL** flags are passed in the *mode* parameter of **sdOpen()**, then a new region will be created if a region with the *name* specified does not already exist in the system. If such a region does exist then no region will be created and **NULL** will be returned.

The behavior of **sdCreate()** is identical to that of **sdOpen()** with both the **OM_CREATE** and **OM_EXCL** flag specified in the *mode* parameter.

While it is possible to specify a physical location of a shared data region with the arguments *physAddress* and *size*, that address range must not be mapped into any other context in the system. No other restrictions are placed. If *physAddress* is **NULL** the system will allocate the physical memory from the available RAM. If there is not enough RAM available in the system the creation will fail and **NULL** will be returned.

It is not possible to specify a virtual location for a shared data region. The location of the region will be returned at *pVirtAddress*.

A *size* of greater than 0 must be specified to create a shared data region.

On creation the shared data region will be mapped into the memory context associated with the task which invoked the call. The shared data region will be owned by either the RTP of that task or the kernel if the task is a kernel task. If the shared data region is owned by a RTP and that RTP exits the kernel will assume ownership of the region.

A shared data region is initially mapped into its owner's context with both read and write access privileges in addition to those specified by *attr*. This may be changed by either a call to **sdProtect()** or **sdMap()**. The MMU attribute value specified in *attr* will be the default value for the shared data region. This will also serve as the limit of access privileges all subsequent clients of the region may use. That is, if *attr* does not specify a particular attribute applications other than the owner will not have, nor be able to set, that attribute on the region within their memory context. For example, if *attr* is set to (**SD_ATTR_RW** | **SD_CACHE_OFF**) an application other than the owner may use **sdProtect()** to restrict its access to (**SD_ATTR_RO** | **SD_CACHE_OFF**), but not to set its access to (**SD_ATTR_RWX** | **SD_CACHE_OFF**).

USING SHARED DATA

To access a shared data region from an application or the kernel it must be initially be mapped to that application via a call to either **sdOpen()** or **sdCreate()**.

These routines return a **SD_ID** which may be used by any task within that application. A **SD_ID** may not be shared between applications or between an application and the kernel.

Once this initial mapping is done tasks in the application may access the memory as if it were local unless explicitly unmapped by a task in the application with a call to **sdUnmap()**.

Task may call the following routines using the application's unique **SD_ID**:

sdDelete()**sdMap()****sdUnmap()****sdProtect()****sdInfoGet()**

By default each client application, excepting the owner, will have the access privileges specified by the value of *attr* at creation. However, an application may change its access privileges via a call to either **sdProtect()** or **sdMap()**, but will be limited to the default attributes of the region or a subset thereof. The owner of a region will by default have both read and write privileges in addition to the region default attributes and may change its local access rights to any valid combination. See **vmBaseLib** for details on what valid values of *attr* are available.

It is important to note that the shared data region object provides no mutual exclusion. If more than one application, or the kernel and one application or more, require access to this region some form of mutual exclusion must be used.

A shared data region may be created that is private to the creator by passing the **SD_PRIVATE** option in the *options* field. No other application, including the kernel, will be able to map such a region.

DELETING SHARED DATA

When all applications have unmapped a shared data region, it may be deleted using the **sdDelete()** function. This will return all resources associated with the region and remove it from the system. It is not possible to delete a shared data region that is still in use by an application or the kernel. To unmap a shared data region from an application it is necessary for a task in that application to call **sdUnmap()**.

By default the last application to unmap a shared data region will force a deletion of the region. However, if the shared data region was created with the option **SD_LINGER** specified it will remain until explicitly deleted by calling **sdDelete()**.

INCLUDE FILES **sdLib.h****SEE ALSO** **rtpLib**, **slLib**, **vmBaseLib**, the VxWorks programmer guides.

sdShow

NAME **sdShow** – Shared Data region show routine**ROUTINES** **sdShow()** – display information for shared data regions

DESCRIPTION	<p>This library provides routines to display information about the Shared Data regions in the system.</p> <p>There are two levels of information that can be obtained: summary and full. For more information see the sdShow() manual entries.</p> <p>The information provided by the show routines should be considered an instantaneous snapshot of the system. This function is only designed as a diagnostic aid. Programmatic access to Shared Data information is provided through the function sdInfoGet(). Refer to the manual entry for this routine for more information.</p> <p>The sdShow() routine may be called only from the C interpreter shell.</p>
CONFIGURATION	To use the shared data region show routine, configure VxWorks with the INCLUDE_SHARED_DATA_SHOW component.
INCLUDE FILES	sdLib.h
SEE ALSO	sdLib , the VxWorks programmer guides.

selectLib

NAME	selectLib – UNIX BSD select library
ROUTINES	<p>selectInit() – initialize the select facility</p> <p>select() – pend on a set of file descriptors</p> <p>selWakeup() – wake up a task pending in select()</p> <p>selWakeupAll() – wake up all tasks in a select() wake-up list</p> <p>selNodeAdd() – add a wake-up node to a select() wake-up list</p> <p>selNodeDelete() – find and delete a node from a select() wake-up list</p> <p>selWakeupListInit() – initialize a select() wake-up list</p> <p>selWakeupListTerm() – terminate a select() wake-up list</p> <p>selWakeupListLen() – get the number of nodes in a select() wake-up list</p> <p>selWakeupType() – get the type of a select() wake-up node</p>
DESCRIPTION	<p>This library provides a BSD 4.3 compatible select facility to wait for activity on a set of file descriptors. selectLib provides a mechanism that gives a driver the ability to detect pended tasks that are awaiting activity on the driver's device. This allows a driver's interrupt service routine to wake up such tasks directly, eliminating the need for polling.</p> <p>Applications can use select() with pipes and serial devices, in addition to sockets. Also, select() examines write file descriptors in addition to read file descriptors; however, exception file descriptors remain unsupported.</p>

Typically, application developers need concern themselves only with the **select()** call. However, driver developers should become familiar with the other routines that may be used with **select()**, if they wish to support the **select()** mechanism.

CONFIGURATION	The select facility is included in a system when VxWorks is configured with the INCLUDE_SELECT component.
INCLUDE FILES	selectLib.h
SEE ALSO	The VxWorks programmer guides.

semBLib

NAME	semBLib – binary semaphore library
ROUTINES	semBInitialize() – initialize a pre-allocated binary semaphore. semBCreate() – create and initialize a binary semaphore
DESCRIPTION	<p>This library provides the interface to VxWorks binary semaphores. Binary semaphores are the most versatile, efficient, and conceptually simple type of semaphore. They can be used to: (1) control mutually exclusive access to shared devices or data structures, or (2) synchronize multiple tasks, or task-level and interrupt-level processes. Binary semaphores form the foundation of numerous VxWorks facilities.</p> <p>A binary semaphore can be viewed as a cell in memory whose contents are in one of two states, full or empty. When a task takes a binary semaphore, using semTake(), subsequent action depends on the state of the semaphore:</p> <ol style="list-style-type: none">(1) If the semaphore is full, the semaphore is made empty, and the calling task continues executing.(2) If the semaphore is empty, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore. <p>When a task gives a binary semaphore, using semGive(), the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore becomes full. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called semGive(), the unblocked task will preempt the calling task.</p>

MUTUAL EXCLUSION

To use a binary semaphore as a means of mutual exclusion, first create it with an initial state of full. For example:

```
SEM_ID semMutex;  
  
/* create a binary semaphore that is initially full */  
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

Then guard a critical section or resource by taking the semaphore with **semTake()**, and exit the section or release the resource by giving the semaphore with **semGive()**. For example:

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

While there is no restriction on the same semaphore being given, taken, or flushed by multiple tasks, it is important to ensure the proper functionality of the mutual-exclusion construct. While there is no danger in any number of processes taking a semaphore, the giving of a semaphore should be more carefully controlled. If a semaphore is given by a task that did not take it, mutual exclusion could be lost.

SYNCHRONIZATION

To use a binary semaphore as a means of synchronization, create it with an initial state of empty. A task blocks by taking a semaphore at a synchronization point, and it remains blocked until the semaphore is given by another task or interrupt service routine.

Synchronization with interrupt service routines is a particularly common need. Binary semaphores can be given, but not taken, from interrupt level. Thus, a task can block at a synchronization point with **semTake()**, and an interrupt service routine can unblock that task with **semGive()**.

In the following example, when **init()** is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. Task 1 will run until it calls **semTake()**, at which point it will block until an event causes the interrupt service routine to call **semGive()**. When the interrupt service routine completes, task 1 can execute to process the event.

```
SEM_ID semSync;    /* ID of sync semaphore */  
  
init ()  
{  
  intConnect (... , eventInterruptSvcRout, ...);  
  semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);  
  taskSpawn (... , task1);  
}  
  
task1 ()  
{  
  ...  
  semTake (semSync, WAIT_FOREVER);    /* wait for event */  
  ... /* process event */
```

```
    }

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

A **semFlush()** on a binary semaphore will atomically unblock all pended tasks in the semaphore queue, i.e., all tasks will be unblocked at once, before any actually execute.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and **intCpuLock** restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are **intCpuLock** restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, **semCLib**, **semMLib**, the VxWorks programmer guides.

semCLib

NAME **semCLib** – counting semaphore library

ROUTINES **semCInitialize()** – initialize a pre-allocated counting semaphore.
 semCCreate() – create and initialize a counting semaphore

DESCRIPTION This library provides the interface to VxWorks counting semaphores. Counting semaphores are useful for guarding multiple instances of a resource.

A counting semaphore may be viewed as a cell in memory whose contents keep track of a count. When a task takes a counting semaphore using **semTake()**, subsequent action depends on the state of the count:

- (1) If the count is non-zero, it is decremented and the calling task continues executing.
- (2) If the count is zero, the task is blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task is removed from the queue of pended tasks and enters the ready state with an **ERROR** status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same counting semaphore.

When a task gives a semaphore, using **semGive()**, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore count is incremented. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called **semGive()**, the unblocked task preempts the calling task.

A **semFlush()** on a counting semaphore atomically unblocks all pended tasks in the semaphore queue. This means all tasks are made ready before any task actually executes. The count of the semaphore remains unchanged.

INTERRUPT USAGE

Counting semaphores may be given but not taken from interrupt level.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores are not given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and **intCpuLock** restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are **intCpuLock** restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib, semBLib, semMLib**

semEvLib

NAME	semEvLib – VxWorks events support for semaphores
ROUTINES	semEvStart() – start the event notification process for a semaphore semEvStop() – stop the event notification process for a semaphore
DESCRIPTION	<p>This library is an extension to eventLib, the events library. Its purpose is to support events for semaphores.</p> <p>The functions in this library are used to control registration of tasks on a semaphore. The routine semEvStart() registers a task and starts the notification process. The function semEvStop() un-registers the task, which stops the notification mechanism.</p> <p>When a task is registered and the semaphore becomes available, the events specified are sent to that task. However, if a semTake() is to be done afterwards, there is no guarantee that the semaphore will still be available.</p>
SMP CONSIDERATIONS	<p>Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling intCpuLock()) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.</p>
INCLUDE FILES	semEvLib.h
SEE ALSO	eventLib , semLib

semExchange

NAME	semExchange – semaphore exchange library
ROUTINES	semExchange() – atomically give and take a pair of semaphores
DESCRIPTION	<p>This library provides the semExchange() routine. This routine atomically gives one semaphore and takes another.</p> <p>Currently on the binary and mutex semaphore types support the semExchange() operation.</p>

The functionality provided by this library can be included/removed from the VxWorks kernel using the **INCLUDE_SEM_EXCHANGE** component.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **semLib.h**

SEE ALSO **taskLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**, **semRWLib**, **semEvLib**, **eventLib**, the VxWorks programmer guides.

semInfo

NAME **semInfo** – semaphore information routines

ROUTINES **semInfo()** – get information about tasks blocked on a semaphore
semInfoGet() – get information about a semaphore

DESCRIPTION This library provides routines to retrieve information about a semaphore.

The routine **semInfo()** returns information about tasks blocked on the semaphore.

Given a **SEM_INFO** structure, the routine **semInfoGet()** returns the type and state of the semaphore, as well as the options used to create the semaphore, the number of blocked tasks and the pending task IDs.

This component is required by the semaphore show routines. It can be included into the VxWorks image using one of the following methods:

Using Workbench

With the kernel configurator include the **INCLUDE_SEM_INFO** component under the **FOLDER_KERNEL** folder.

Using the vxprj Command Line Tool

Use the *add* command to include the **INCLUDE_SEM_INFO** component.

INCLUDE FILES **semLib.h**

semLib

NAME	semLib – general semaphore library
ROUTINES	semGive() – give a semaphore semTake() – take a semaphore semFlush() – unblock every task pended on a semaphore semDelete() – delete a semaphore
DESCRIPTION	<p>Semaphores are the basis for synchronization and mutual exclusion in VxWorks. They are powerful in their simplicity and form the foundation for numerous VxWorks facilities.</p> <p>Different semaphore types serve different needs, and while the behavior of the types differs, their basic interface is the same. This library provides semaphore routines common to all VxWorks semaphore types. For all types, the two basic operations are semTake() and semGive(), the acquisition or relinquishing of a semaphore.</p> <p>Semaphore creation and initialization is handled by other libraries, depending on the type of semaphore used. These libraries contain full functional descriptions of the semaphore types:</p> <ul style="list-style-type: none">semBLib - binary semaphoressemCLib - counting semaphoressemMLib - mutual exclusion semaphoressemRWLib - reader/writer semaphoressemSmLib - shared memory semaphores <p>Binary semaphores offer the greatest speed and the broadest applicability.</p> <p>The semLib library provides all other semaphore operations, including routines for semaphore control, deletion, and information. Semaphores must be validated before any semaphore operation can be undertaken. An invalid semaphore ID results in ERROR, and an appropriate errno is set.</p>

SEMAPHORE CONTROL

The **semTake()** call acquires a specified semaphore, blocking the calling task or making the semaphore unavailable. All semaphore types support a timeout on the **semTake()** operation. The timeout is specified as the number of ticks to remain blocked on the semaphore. Timeouts of **WAIT_FOREVER** and **NO_WAIT** codify common timeouts. If a **semTake()** times out, it returns **ERROR**. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The **semGive()** call relinquishes a specified semaphore, unblocking a pended task or making the semaphore available. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The **semFlush()** call may be used to atomically unblock all tasks pended on a semaphore queue, i.e., all tasks will be unblocked before any are allowed to run. It may be thought of as a broadcast operation in synchronization applications. The state of the semaphore is unchanged by the use of **semFlush()**; it is not analogous to **semGive()**.

SEMAPHORE DELETION

The **semDelete()** call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return **ERROR**. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that already has taken (owns) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully taken.

SEMAPHORE INFORMATION

The **semInfo()** call is a useful debugging aid, reporting all tasks blocked on a specified semaphore. It provides a snapshot of the queue at the time of the call, but because semaphores are dynamic, the information may be out of date by the time it is available. As with the current state of the semaphore, use of the queue of pended tasks should be restricted to debugging uses only.

VXWORKS EVENTS If a task has registered for receiving events with a semaphore, events will be sent when that semaphore becomes available. By becoming available, it is implied that there is a change of state. For a binary semaphore, there is only a change of state when a **semGive()** is done on a semaphore that was taken. For a counting semaphore, there is always a change of state when the semaphore is available, since the count is incremented each time. For a mutex, a **semGive()** can only be performed if the current task is the owner, implying that the semaphore has been taken; thus, there is always a change of state. Events are not currently supported for use with reader/writer semaphores.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **semLib.h**

SEE ALSO **taskLib, semBLib, semCLib, semMLib, semSmLib, semRWLib, semEvLib, eventLib**, the VxWorks programmer guides.

semMLib

NAME	semMLib – mutual-exclusion semaphore library
ROUTINES	semMInitialize() – initialize a pre-allocated mutex semaphore. semMGiveForce() – give a mutual-exclusion semaphore without restrictions semMCreate() – create and initialize a mutual-exclusion semaphore
DESCRIPTION	<p>This library provides the interface to VxWorks mutual-exclusion semaphores. Mutual-exclusion semaphores offer convenient options suited for situations requiring mutually exclusive access to resources. Typical applications include sharing devices and protecting data structures. Mutual-exclusion semaphores are used by many higher-level VxWorks facilities.</p> <p>The mutual-exclusion semaphore is a specialized version of the binary semaphore, designed to address issues inherent in mutual exclusion, such as recursive access to resources, priority inversion, and deletion safety. The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore (see the manual entry for semBLib), except for the following restrictions:</p> <ul style="list-style-type: none">- It can only be used for mutual exclusion.- It can only be given by the task that took it.- It may not be taken or given from interrupt level.- The semFlush() operation is illegal.

These last two operations have no meaning in mutual-exclusion situations.

RECURSIVE RESOURCE ACCESS

A special feature of the mutual-exclusion semaphore is that it may be taken "recursively," i.e., it can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other.

Recursion is possible because the system keeps track of which task currently owns a mutual-exclusion semaphore. Before being released, a mutual-exclusion semaphore taken recursively must be given the same number of times it has been taken; this is tracked by means of a count which is incremented with each **semTake()** and decremented with each **semGive()**.

The example below illustrates recursive use of a mutual-exclusion semaphore. Function A requires access to a resource which it acquires by taking **semM**; function A may also need to call function B, which also requires **semM**:

```
SEM_ID semM;  
  
semM = semMCreate (...);
```

```
funcA ()
{
    semTake (semM, WAIT_FOREVER);
    ...
    funcB ();
    ...
    semGive (semM);
}

funcB ()
{
    semTake (semM, WAIT_FOREVER);
    ...
    semGive (semM);
}
```

PRIORITY-INVERSION SAFETY

If the option **SEM_INVERSION_SAFE** is selected, the library adopts a priority-inheritance protocol to resolve potential occurrences of "priority inversion," a problem stemming from the use semaphores for mutual exclusion. Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task.

Consider the following scenario: T1, T2, and T3 are tasks of high, medium, and low priority, respectively. T3 has acquired some resource by taking its associated semaphore. When T1 preempts T3 and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that T1 would be blocked no longer than the time it normally takes T3 to finish with the resource, the situation would not be problematic. However, the low-priority task is vulnerable to preemption by medium-priority tasks; a preempting task, T2, could inhibit T3 from relinquishing the resource. This condition could persist, blocking T1 for an indefinite period of time.

The priority-inheritance protocol solves the problem of priority inversion by elevating the priority of T3 to the priority of T1 during the time T1 is blocked on T3. This protects T3, and indirectly T1, from preemption by T2. Stated more generally, the priority-inheritance protocol assures that a task which owns a resource will execute 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 contributing mutual-exclusion semaphores that the task owns 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.

The priority-inheritance protocol also takes into consideration a task's ownership of more than one mutual-exclusion semaphore at a time. Such a task will execute at the priority of the highest priority task blocked on any of its owned resources. Under most circumstances, the task will return to its normal priority only after relinquishing all contributing mutual-exclusion semaphores.

The sole exception to this occurs if some task tried to lower the priority of a task involved in priority inheritance by using **taskPrioritySet()**. This act creates some uncertainties in that task's inheritance tracking that can only be made certain when all the inversion safe mutual-exclusion semaphores that task has are known to be involved in its priority

inheritance. If all previously known contributing mutual-exclusion semaphores were to be relinquished, the priority of that task might not be lowered to its newly assigned lower priority. It will however at least be lowered to the last known "safe" value-- typically the task's normal priority before that call to **taskPrioritySet()**. The priority can not be restored to the new normal priority before these uncertainties are removed. At the absolute worst, this is not until the task gives up all of its inversion safe mutual-exclusion semaphores.

SEMAPHORE DELETION

The **semDelete()** call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return **ERROR**. Take special care when deleting mutual-exclusion semaphores to avoid deleting a semaphore out from under a task that already owns (has taken) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task owns.

TASK-DELETION SAFETY

If the option **SEM_DELETE_SAFE** is selected, the task owning the semaphore will be protected from deletion as long as it owns the semaphore. This solves another problem endemic to mutual exclusion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource would be unavailable, effectively shutting off all access to the resource.

As discussed in **taskLib**, the primitives **taskSafe()** and **taskUnsafe()** offer one solution, but as this type of protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore provides the option **SEM_DELETE_SAFE**, which enables an implicit **taskSafe()** with each **semTake()**, and a **taskUnsafe()** with each **semGive()**. This convenience is also more efficient, as the resulting code requires fewer entrances to the kernel.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The **SEM_DELETE_SAFE** option partially protects an application, to the extent that unexpected deletions will be deferred until the resource is released.

Because the priority of a task which has been elevated by the taking of a mutual-exclusion semaphore remains at the higher priority until all mutexes held by that task are released, unbounded priority inversion situations can result when nested mutexes are involved. If nested mutexes are required, consider the following alternatives:

1. Avoid overlapping critical regions.
2. Adjust priorities of tasks so that there are no tasks at intermediate priority levels.
3. Adjust priorities of tasks so that priority inheritance protocol is not needed.

- 4. Manually implement a static priority ceiling protocol using a non-inversion-save mutex. This involves setting all blockers on a mutex to the ceiling priority, then taking the mutex. After semGive, set the priorities back to the base priority. Note that this implementation reduces the queue to a fifo queue.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, **semBLib**, **semCLib**, the VxWorks programmer guides.

semOpen

NAME **semOpen** – extended semaphore library

ROUTINES **semOpenInit()** – initialize the semaphore open facility
semOpen() – open a named semaphore
semClose() – close a named semaphore
semUnlink() – unlink a named semaphore

DESCRIPTION The extended semaphore library includes the APIs to open, close, and unlink semaphores. Since these APIs did not exist in VxWorks 5.5, to prevent the functions from being included in the default image, they have been isolated from the general semaphore library.

INCLUDE FILES **semLib.h**

SEE ALSO **msgQOpen**, **objOpen**, **taskOpen**, **timerOpen**, the VxWorks, programmer guides.

semPxLib

NAME **semPxLib** – semaphore synchronization library (POSIX)

ROUTINES	<p>semPxBLibInit() – initialize POSIX semaphore support</p> <p>sem_init() – initialize an unnamed semaphore (POSIX)</p> <p>sem_destroy() – destroy an unnamed semaphore (POSIX)</p> <p>sem_open() – initialize/open a named semaphore (POSIX)</p> <p>sem_close() – close a named semaphore (POSIX)</p> <p>sem_unlink() – remove a named semaphore (POSIX)</p> <p>sem_wait() – lock (take) a semaphore, blocking if not available (POSIX)</p> <p>sem_trywait() – lock (take) a semaphore, returning error if unavailable (POSIX)</p> <p>sem_timedwait() – lock (take) a semaphore with a timeout (POSIX)</p> <p>sem_post() – unlock (give) a semaphore (POSIX)</p> <p>sem_getvalue() – get the value of a semaphore (POSIX)</p>
DESCRIPTION	<p>This library implements the semaphore interface based on the POSIX 1003.1b specifications. For alternative semaphore routines designed expressly for VxWorks, see the manual page for semLib and other semaphore libraries mentioned there. POSIX semaphores are counting semaphores; as such they are most similar to the semCLib VxWorks-specific semaphores.</p> <p>The main advantage of POSIX semaphores is portability (to the extent that alternative operating systems also provide these POSIX interfaces). However, VxWorks-specific semaphores provide the following features absent from the semaphores implemented in this library: priority inheritance, task-deletion safety, the ability for a single task to take a semaphore multiple times, ownership of mutual-exclusion semaphores, semaphore timeout, and the choice of queuing mechanism.</p> <p>POSIX defines both named and unnamed semaphores; semPxBLib includes separate routines for creating and deleting each kind. For other operations, applications use the same routines for both kinds of semaphore.</p>
TERMINOLOGY	<p>The POSIX standard uses the terms <i>wait</i> or <i>lock</i> where <i>take</i> is normally used in VxWorks, and the terms <i>post</i> or <i>unlock</i> where <i>give</i> is normally used in VxWorks. VxWorks documentation that is specific to the POSIX interfaces (such as the remainder of this manual entry, and the manual entries for subroutines in this library) uses the POSIX terminology, in order to make it easier to read in conjunction with other references on POSIX.</p>
SEMAPHORE DELETION	<p>The sem_destroy() call terminates an unnamed semaphore and deallocates any associated memory; the combination of sem_close() and sem_unlink() has the same effect for named semaphores. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that has already locked that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully locked. (Similarly, for named semaphores, applications should take care to only close semaphores that the closing task has opened.)</p> <p>If there are tasks blocked waiting for the semaphore, sem_destroy() fails and sets errno to EBUSY.</p>

Detection of deadlock is not considered in this implementation.

INCLUDE FILES	semaphore.h
SEE ALSO	POSIX 1003.1b document, semLib , the VxWorks programmer guides.

semPxShow

NAME	semPxShow – POSIX semaphore show library
ROUTINES	semPxShowInit() – initialize the POSIX semaphore show facility semPxShow() – display semaphore internals
DESCRIPTION	This library provides a show routine for POSIX semaphore objects.
INCLUDE FILES	semPxShow.h

semRWLib

NAME	semRWLib – reader/writer semaphore library
ROUTINES	semRWInitialize() – initialize a pre-allocated read/write semaphore. semWTake() – take a semaphore in write mode semRTake() – take a semaphore as a reader semRWGiveForce() – give a reader/writer semaphore without restrictions semRWCreate() – create and initialize a reader/writer semaphore
DESCRIPTION	This library provides the interface to VxWorks reader/writer semaphores. Reader/writer semaphores provide a method of synchronizing groups of tasks that can be granted concurrent access to a resource with those tasks that require mutually exclusive access to that resource. Typically this correlates to those tasks that intend to modify a resource and those which intend only to view it.

Like a mutual-exclusion semaphore the following restrictions exist:

- It can only be given by the task that took it.
- It may not be taken or given from interrupt level.
- The **semFlush()** operation is illegal.

A reader/writer semaphore differs from other semaphore types in that a mode is specified by the choice of the "take" routine. It is this mode that determines whether the caller requires mutually exclusive access or if concurrent access would suffice.

The two modes are "read" and "write", and specified by calling one of the following routines:

semRTake() - take a semaphore in "read" mode

semWTake() - take a semaphore in "write" mode

For tasks that take a reader/writer semaphore in "write" mode the behavior is quite similar to a mutex semaphore. That task will own the semaphore exclusively.

If a timeout other than **NO_WAIT** is specified an attempt to acquire a reader/writer semaphore in "write" mode when the semaphore is held by another writer or any number of readers will result in the caller pending.

The behavior of a reader/writer semaphore when taken in "read" mode is unique. This does not imply exclusive access to a resource. In fact, a semaphore may be concurrently held in this mode by a number of tasks. These tasks can be seen as collectively owning the semaphore.

Mutual exclusion between a collection of reader tasks and all writer tasks will be maintained.

If a timeout other than **NO_WAIT** is specified an attempt to acquire a reader/writer semaphore in "read" mode when the semaphore is held by a writer will result in the caller pending. Also, if the semaphore is held by other readers but the maximum concurrent readers has been reached the caller will pend. If a task has attempted to take the semaphore in "write" mode and pended for any reason all subsequent "read" takes will result in the caller pending until all writers have run.

When a reader/writer semaphore becomes available a new owner is selected from any tasks pended on the semaphore. If tasks are pended in "write" mode they will be granted ownership in the order determined by the option specified for the semaphore at creation (**SEM_Q_FIFO** or **SEM_Q_PRIORITY**). If no write tasks are pended then all tasks waiting for the semaphore in "read" mode, up to the maximum concurrent readers specified for the semaphore, will be granted ownership in "read" mode.

Though the maximum number of concurrent readers is set per semaphore at creation there is also a limit on the maximum concurrent readers for a system as defined by

SEM_RW_MAX_CONCURRENT_READERS. The value of **SEM_RW_MAX_CONCURRENT_READERS** will be used as the semaphore's maximum if a larger value is specified at creation. This value should be set no larger than necessary as a larger maximum concurrent reader value will result in longer interrupt and task response.

RECURSIVE RESOURCE ACCESS

Like mutex semaphores reader/writer semaphores support recursive access. Please refer to the **semMLib** documentation for further details.

WARNING While taking a reader/writer semaphore recursively through either the `semWTake` and `semRTake` routines is allowed, an attempt to acquire a semaphore in both modes is not allowed. The `semWTake()` routine will return **ERROR** if the semaphore is held by the caller as a reader and the `semRTake()` routine will return **ERROR** if the semaphore is held by the caller as a writer.

PRIORITY-INVERSION SAFETY

Like mutex semaphores reader/writer semaphores support priority inheritance. Please refer to the **semMLib** documentation for further details.

SEMAPHORE DELETION

The `semDelete()` call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return **ERROR**. Take special care when deleting read/write semaphores to avoid deleting a semaphore out from under tasks that have taken that semaphore. In particular, a semaphore should never be deleted when held in read mode and the option `SEM_DELETE_SAFE` was passed at creation.

Applications should adopt the protocol of only deleting semaphores that the deleting task owns in write mode.

TASK-DELETION SAFETY

Like mutex semaphores reader/writer semaphores support task deletion safety. Please refer to the **semMLib** documentation for further details.

INCLUDE FILES `semLib.h`

SEE ALSO `semLib`, `semMLib`, `semBLib`, `semCLib`, VxWorks Programmer's Guide

semShow

NAME `semShow` – semaphore show routines

ROUTINES `semShow()` – show information about a semaphore

DESCRIPTION This library provides routines to show semaphore statistics, such as semaphore type, semaphore queuing method, tasks pended, etc.

The semaphore show facility is configured into VxWorks using either of the following methods:

Using Workbench

With the kernel configurator include the `INCLUDE_SEM_SHOW` component under the `FOLDER_SHOW_ROUTINES` folder.

Using the vxprj Command Line Tool

Use the `add` command to include the `INCLUDE_SEM_SHOW` component.

Routines in this library are meant to be used as debugging aids that display semaphore information to standard output. Due to the dynamic nature of semaphore operations the information displayed may no longer be accurate by the time it is provided.

INCLUDE FILES	semLib.h
SEE ALSO	semLib , <i>VxWorks Programmer's Guide</i>

semSmLib

NAME	semSmLib – shared memory semaphore library (VxMP Option)
ROUTINES	<p>semBSmCreate() – create and initialize a shared memory binary semaphore (VxMP Option)</p> <p>semCSmCreate() – create and initialize a shared memory counting semaphore (VxMP Option)</p>
DESCRIPTION	<p>This library provides the interface to VxWorks shared memory binary and counting semaphores. Once a shared memory semaphore is created, the generic semaphore-handling routines provided in semLib are used to manipulate it. Shared memory binary semaphores are created using semBSmCreate(). Shared memory counting semaphores are created using semCSmCreate().</p> <p>Shared memory binary semaphores are used to: (1) control mutually exclusive access to multiprocessor-shared data structures, or (2) synchronize multiple tasks running in a multiprocessor system. For general information about binary semaphores, see the manual entry semBLib.</p> <p>Shared memory counting semaphores are used for guarding multiple instances of a resource used by multiple CPUs. For general information about shared counting semaphores, see the manual entry for semCLib.</p> <p>For information about the generic semaphore-handling routines, see the manual entry for semLib.</p>
MEMORY REQUIREMENTS	The semaphore structure is allocated from a dedicated shared memory partition.

The shared semaphore dedicated shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared semaphores, set in the configuration parameter **SM_OBJ_MAX_SEM**.

This memory partition is common to shared binary and counting semaphores, thus **SM_OBJ_MAX_SEM** must be set to the sum total of binary and counting semaphores to be used in the system.

RESTRICTIONS	Shared memory semaphores differ from local semaphores in the following ways: Interrupt Use: Shared semaphores may not be given, taken, or flushed at interrupt level. Deletion: There is no way to delete a shared semaphore and free its associated shared memory. Attempts to delete a shared semaphore return ERROR and set errno to S_smObjLib_NO_OBJECT_DESTROY . Queuing Style: The shared semaphore queuing style specified when the semaphore is created must be FIFO.
---------------------	--

INTERRUPT LATENCY	Internally, interrupts are locked while manipulating shared semaphore data structures, thus increasing local CPU interrupt latency.
CONFIGURATION	Before routines in this library can be called, the shared memory object facility must be initialized by calling usrSmObjInit() . This is done automatically during VxWorks initialization when the component INCLUDE_SM_OBJ is included.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory support option, VxMP.
INCLUDE FILES	semSmLib.h
SEE ALSO	semLib , semBLib , semCLib , smObjLib , semShow , usrSmObjInit() , the VxWorks programmer guides.

shellConfigLib

NAME	shellConfigLib – the shell configuration management module
ROUTINES	shellConfigDefaultSet() – set default shell configuration shellConfigSet() – set shell configuration

shellConfigDefaultGet() – get default shell configuration
shellConfigGet() – get the shell configuration
shellConfigDefaultValueSet() – set a default configuration variable value
shellConfigValueSet() – set a shell configuration variable value
shellConfigDefaultValueUnset() – unset a default configuration variable value
shellConfigValueUnset() – unset a shell configuration variable value
shellConfigDefaultValueGet() – get a default configuration variable value
shellConfigValueGet() – get a shell configuration variable value

DESCRIPTION	<p>This module manages the configuration variables of the kernel shell.</p> <p>Configuration variables are used to store dynamic configuration of the shell core mechanism itself or of shell commands. This concept is very similar to the UNIX or Windows shell environment variables.</p> <p>Configuration variables can be set either globally to all shell sessions, or locally to a specific one. A local configuration variable supersedes the global definition, if one exists.</p> <p>It is also possible to unset a configuration variable (global or local).</p>
INCLUDE FILES	shellConfigLib.h, shellLib.h
SEE ALSO	shellLib , <i>VxWorks Kernel Programmer's Guide: Kernel Shell</i>

shellDataLib

NAME	shellDataLib – the shell data management module
ROUTINES	<p>shellDataFromNameAdd() – add user data to a specified shell</p> <p>shellDataAdd() – add user data to a specified shell</p> <p>shellDataRemove() – remove user data from a specified shell</p> <p>shellDataFromNameGet() – get user data from a specified shell</p> <p>shellDataGet() – get user data from a specified shell</p> <p>shellDataFirst() – get the first user data that matches a key</p> <p>shellDataNext() – get the next user data that matches a key</p>
DESCRIPTION	<p>This module manages the user data that can be stored in or retrieved from a shell session context.</p> <p>This facility allows a routine called from a shell session to store private or public information for a latter use by itself or an other routine. This information is local to a specified shell session.</p>

Information are stored and retrieved using a key string that should be unique for a dedicated information. Along with that key, an integer value is stored that defines the data value. This value can be a pointer on specific structure for example.

At the shell termination, the data value may need a special handling. For example if this is a pointer to an allocated buffer, it has to be freed in order to prevent memory leaks. For that purpose, when a *data* is added to a shell session context, it is possible to specify a finalizing routine. This routine will be called automatically when the shell session is terminated.

INCLUDE FILES **shellDataLib.h**

SEE ALSO **shellLib**, *VxWorks Kernel Programmer's Guide: Kernel Shell*

shellInterpCmdLib

NAME **shellInterpCmdLib** – the command interpreter library

ROUTINES **shellCmdPreParseAdd()** – define a command to be pre-parsed
shellCmdMemRegister() – register a buffer against the command interpreter
shellCmdMemUnregister() – unregister a buffer
shellCmdExec() – execute a shell command
shellCmdAdd() – add a shell command
shellCmdArrayAdd() – add an array of shell commands
shellCmdTopicAdd() – add a shell command topic
shellCmdAliasAdd() – add an alias string
shellCmdAliasArrayAdd() – add an array of alias strings
shellCmdAliasDelete() – delete an alias
shellCmdSymTabIdGet() – get symbol table Id of a shell session

DESCRIPTION This module contains several routines to manages the shell commands for the shell command interpreter. At initialization time, it registers several commands in order to access the tasks, the symbols, the file system, the memory, the breakpoints, the network and the object. Some other basic commands are also registered. Check the command manual for a list of available commands.

The module exports the necessary routines in order for the customer to create and easily add its own commands to the command interpreter.

INCLUDE FILES **shellInterpCmdLib.h**

shellInterpLib

NAME	shellInterpLib – the shell interpreters management module
ROUTINES	shellInterpEvaluate() – interpret a string by an interpreter shellInterpRegister() – register a new interpreter shellInterpByNameFind() – Find an interpreter based on its name shellInterpCtxGet() – get the interpreter context shellInterpDefaultNameGet() – get the name of the default interpreter shellInterpNameGet() – get the name of the current interpreter
DESCRIPTION	This module manages the multiple interpreter capability of the kernel shell. Few routines are available from this library, mainly to register an interpreter, get the name of the current interpreter of the shell session and get the current interpreter context of a shell session.
INCLUDE FILES	shellInterpLib.h
SEE ALSO	shellLib , <i>VxWorks Kernel Programmer's Guide: Kernel Shell</i>

shellLib

NAME	shellLib – the kernel shell module
ROUTINES	shellGenericInit() – start a shell session shellRestart() – restart a shell session shellAbort() – abort a shell session shellPromptSet() – change the shell prompt (vxWorks 5.5 compatibility) shellScriptAbort() – signal the shell to stop processing a script (vxWorks 5.5 compatibility) shellHistory() – display or set the size of the shell history (vxWorks 5.5 compatibility) shellLock() – lock access to the shell (vxWorks 5.5 compatibility) shellFirst() – get the first shell session shellNext() – get the next shell session shellIdVerify() – verify the validity of a shell session Id shellTaskGet() – get the task Id of a shell session shellFromTaskGet() – get a shell session Id from its task Id shellFromNameGet() – get a shell session Id from a task name shellErrnoSet() – set the shell session errno shellErrnoGet() – get the shell session errno shellCompatibleCheck() – check the compatibility mode of the shell shellTerminate() – terminate a shell task shellTaskIdDefault() – set the default task for a given shell session

shellResourceReleaseHookAdd() – add a resource-releasing hook to the shell

DESCRIPTION	<p>This library contains the execution support routines for the VxWorks kernel shell. The kernel shell provides the basic programmer's interface to VxWorks.</p> <p>This module gives access to the kernel shell. It is used to launch a new shell task (shellXXXInit() functions), to control it (shellRestart()) and to end it (shellTerminate()).</p>
INTERPRETERS	<p>The kernel shell is based on different interpreters:</p> <ul style="list-style-type: none">- A C-expression interpreter, containing no built-in commands (as in previous version of VxWorks).- A command interpreter, containing several commands to manipulate task, file systems, network, objects, symbols... This interpreter is very similar to a UNIX shell interpreter like sh or csh. The customer can add his/her own command to that interpreter.- Any other interpreter of the customer. <p>The interpreters are registered against the shell at boot time (see the shellInterpLib library). It is possible to switch dynamically from one interpreter to another using either a dedicated command or by modifying directly the shell configuration variable INTERPRETER (see below).</p> <p>Each interpreter has its own private line history, and so the interpreter syntax are not mixed.</p> <p>The nature, use, and syntax of the shell are fully described in the <i>VxWorks Kernel Programmer's Guide: Kernel Shell</i> and <i>Wind River Workbench Command-Line User's Guide 2.2: Host Shell</i>.</p>

MULTIPLE SHELL SESSIONS

More than one kernel shell session can be launched at a time. But only one shell task can have its standard input streams (**STD_IN**) attached to the console. Each of the shell tasks is characterized by a shell session identifier (type **SHELL_ID**) that can be known either by the shell task name (**shellFromNameGet()**) or by the shell task ID (**shellFromTaskGet()**). Each shell session has normally its own set of I/O, that does not interfere with the other shell sessions or with the system. In that case, the global standard I/O is not modified by the shell. Moreover, modifying the global standard I/O will not modify the I/O of the shell session attached to the console. Dedicated APIs exist to modify a shell session I/O.

Obviously, this new feature may break the compatibility with previous version of VxWorks. For that reason, it is possible to turn that feature off at kernel configuration time (check the compatibility parameter of the kernel shell component). Turning this feature off sets the kernel shell to a compatibility mode: one shell session shared between the different connections (telnet/rlogin/console/wtxConsole) that modifies the global standard I/O of the system.

SECURE ACCESS	As for previous version of the shell, a remote connection is secured by a login/password step if the security component is included into the VxWorks image. The secure access has
----------------------	---

been extended to any other shell session, including the session attached to the console. This feature is enabled by the `secure` parameter of the shell.

Two routines exist to set the login and logout function that have to be called by the shell. They have to be set before creating a shell session. They are common to all shell session initialization processes, but are copied into the shell context. As a consequence, when a shell session is started, it is not possible to modify those functions anymore.

CONFIGURATION VARIABLES

Some behavior of the kernel shell can be modified through the use of configuration variables. These variables can be defined at boot time (see the configuration parameters of the shell component) and dynamically created and modified (see either dedicated command of the interpreter or the **shellConfigLib** library). Configuration variables are like UNIX shell variables or UNIX environment variables.

There are global variables (common to all shell sessions) and local variables to one shell session. If one creates a local variable with the same name as a global variable, the local overloads the global. If one modifies the value of a global variable with a routine dedicated to local variables (see **shellConfigLib**), a new local variable with the same name is created. Most of the routines or commands have a local scope, and so do not modify the configuration values globally.

The shell defines and uses several configuration variables. Other modules may define and use new variables.

The existing configuration variables used the core mechanism of the shell are:

INTERPRETER

Define the name of the current interpreter. Setting this variable allows to switch from one interpreter to another one. The existing interpreters that come along the shell is the C interpreter named "C" and the command interpreter named "Cmd".

LINE_LENGTH

Set the line editing length value. The value is taken into account only when a new session is created.

LINE_EDIT_MODE

Define the name of the line edit mode to use for the shell session. The existing line editing modes that come along the shell is "vi" and "emacs".

EXC_PRINT

If this variable is set to "on" (default), the exceptions are reported to the shell session. Setting it to "off" stops the exception reporting to the shell session.

BP_PRINT

If this variable is set to "on" (default), the breakpoint notifications are reported to the shell session. Setting it to "off" stops the breakpoint notifications reporting to the shell session.

AUTOLOGOUT

Set the autologout delay, in minutes. This variable is used when the shell session is accessed with a login/password. After the logout delay, and if no character is type to the shell terminal, the shell session is automatically log out.

CPLUS_SYM_MATCH

When the kernel shell try to access a symbol, it first looks for its name as a C symbol name. If the symbol cannot be located, the shell looks for its name as a partial C++ mangled symbol name. If this variable is set to "on" (default is "off"), the shell always search for both symbol name format.

LINE EDITING MODE

The kernel shell has two line editing modes: the classical Vi-like one and the new Emacs-like one (see **ledLib** documentation). Each and both can be added to the VxWorks kernel. If both editing modes are included into the VxWorks image, it is possible to dynamically switch between them, using the configuration variable **LINE_EDIT_MODE**.

SHELL TASK STACK SIZE

The kernel shell is using the VxWorks demangler to access C++ symbol names. The demangler implementation uses a recursive descent parsing algorithm to decode C++ mangled symbol names. Demangling of symbols with an extremely high degree of template nesting can require an arbitrarily large amount of stack space (nesting level of ~800 requires ~80K of stack), simply because of the depth of the call stack. If such symbols should be accessed from the shell, it may be necessary to increase the shell task stack size, using the shell component parameter **SHELL_STACK_SIZE**.

Notice that C++ standard says that implementations are not required to support a template nesting depth greater than 17.

INCLUDE FILES **shellLib.h**

SEE ALSO *VxWorks Kernel Programmer's Guide: Kernel Shell, Wind River Workbench Command-Line User's Guide 2.2: Host Shell*

shellPromptLib

NAME **shellPromptLib** – the shell prompt management module

ROUTINES **shellPromptFmtStrAdd()** – add a new prompt format string
shellPromptFmtSet() – set the current prompt format string
shellPromptFmtDftSet() – set the default prompt format string

DESCRIPTION	<p>This module manages the shell prompt strings defined by the interpreters. The shell prompt is a regular string that can contain format strings to print various information (as for a UNIX shell).</p> <p>The format strings are composed of the percent (%) character plus one character. This library provides the format strings:</p> <p>%/ display the current working path</p> <p>%h display the current history event number</p> <p>%m display the target name</p> <p>%% display the percent character</p> <p>%n display the current user name</p> <p>Other modules of the kernel can add their own format string using the function shellPromptFmtStrAdd().</p>
INCLUDE FILES	shellLib.h
SEE ALSO	shellLib, <i>VxWorks Kernel Programmer's Guide: Kernel Shell</i>

shlShow

NAME	shlShow – Shared Library Show Routine
ROUTINES	shlShow() – display information for shared libraries rtpShlShow() – Display shared library information for an RTP
DESCRIPTION	<p>This library provides routines to display information about the Shared libraries in the Real Time Processes (RTP). The routines are only included if the shared library component (INCLUDE_SHL) and the shl show component (INCLUDE_SHL_SHOW) are configured into the kernel.</p> <p>The information provided by the show routines should be considered an instantaneous snapshot of the system. The show function is designed only as a diagnostic aid and should not be used programmatically.</p> <p>The shlShow() routine is called from the C interpreter shell.</p>
INCLUDE FILES	shlLib.h
SEE ALSO	shlLib, rtpLib, the VxWorks programmer guides.

sigLib

NAME	sigLib – software signal facility library
ROUTINES	<p>sigInit() – initialize the signal facilities</p> <p>sigqueueInit() – initialize the queued signal facilities</p> <p>sigemptyset() – initialize a signal set with no signals included (POSIX)</p> <p>sigfillset() – initialize a signal set with all signals included (POSIX)</p> <p>sigaddset() – add a signal to a signal set (POSIX)</p> <p>sigdelset() – delete a signal from a signal set (POSIX)</p> <p>sigismember() – test to see if a signal is in a signal set (POSIX)</p> <p>signal() – specify the handler associated with a signal</p> <p>sigaction() – examine and/or specify the action associated with a signal (POSIX)</p> <p>sigprocmask() – examine and/or change the signal mask (POSIX)</p> <p>sigpending() – retrieve the set of pending signals blocked from delivery (POSIX)</p> <p>sigsuspend() – suspend the task until delivery of a signal (POSIX)</p> <p>pause() – suspend the task until delivery of a signal (POSIX)</p> <p>sigtimedwait() – wait for a signal</p> <p>sigwaitinfo() – wait for real-time signals</p> <p>sigwait() – wait for a signal to be delivered (POSIX)</p> <p>sigvec() – install a signal handler</p> <p>sigsetmask() – set the signal mask</p> <p>sigblock() – add to a set of blocked signals</p> <p>raise() – send a signal to the caller's task</p> <p>taskRaise() – send a signal to the caller's task</p> <p>kill() – send a signal to a task (POSIX)</p> <p>taskKill() – send a signal to a task</p> <p>sigqueue() – send a queued signal to a task</p> <p>taskSigqueue() – send a queued signal to a task</p>
DESCRIPTION	<p>This library provides a signal interface for tasks. Signals are used to alter the flow control of tasks by communicating asynchronous events within or between task contexts. Any task or interrupt service can "raise" (or send) a signal to a particular task. The task being signaled will immediately suspend its current thread of execution and invoke a task-specified "signal handler" routine. The signal handler is a user-supplied routine that is bound to a specific signal and performs whatever actions are necessary whenever the signal is received. Signals are most appropriate for error and exception handling, rather than as a general purpose intertask communication mechanism.</p> <p>This library has both a BSD 4.3 and POSIX signal interface. The POSIX interface provides a standardized interface which is more functional than the traditional BSD 4.3 interface. The chart below shows the correlation between BSD 4.3 and POSIX 1003.1 functions. An application should use only one form of interface and not intermix them.</p>

BSD 4.3	POSIX 1003.1
sigmask()	sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember()
sigblock()	sigprocmask()
sigsetmask()	sigprocmask()
pause()	sigsuspend()
sigvec()	sigaction()
(none)	sigpending()
signal()	signal()
kill()	kill()

POSIX 1003.1b (Real-Time Extensions) also specifies a queued-signal facility that involves four additional routines: **sigqueue()**, **sigwaitinfo()**, and **sigtimedwait()**.

The default handling of a signal differs significantly from the POSIX specification. When **SIG_DFL** is specified as the value of its handler a signal will be ignored.

In many ways, signals are analogous to hardware interrupts. The signal facility provides a set of 63 distinct signals. A signal can be raised by calling **kill()**, which is analogous to an interrupt or hardware exception. A signal handler is bound to a particular signal with **sigaction()** in much the same way that an interrupt service routine is connected to an interrupt vector with **intConnect()**. Signals are blocked for the duration of the signal handler, just as interrupts are locked out for the duration of the interrupt service routine. Tasks can block the occurrence of certain signals with **sigprocmask()**, just as the interrupt level can be raised or lowered to block out levels of interrupts. If a signal is blocked when it is raised, its handler routine will be called when the signal becomes unblocked. Caution is suggested when calling routines that may block during a signal handler as this may introduce deadlock situations.

Several routines (**sigprocmask()**, **sigpending()**, and **sigsuspend()**) take **sigset_t** data structures as parameters. These data structures are used to specify signal set masks. Several routines are provided for manipulating these data structures: **sigemptyset()** clears all the bits in a **sigset_t**, **sigfillset()** sets all the bits in a **sigset_t**, **sigaddset()** sets the bit in a **sigset_t** corresponding to a particular signal number, **sigdelset()** resets the bit in a **sigset_t** corresponding to a particular signal number, and **sigismember()** tests to see if the bit corresponding to a particular signal number is set.

CONFIGURATION To use the software signal facility library, configure VxWorks with the **INCLUDE_SIGNALS** component.

FUNCTION RESTARTING

If a task is pended (for instance, by waiting for a semaphore to become available) and a signal is sent to the task for which the task has a handler installed, then the handler will run before the semaphore is taken. When the handler returns the task will go back to being

pended (waiting for the semaphore). If there was a timeout used for the pend, then the original value will be used again when the task returns from the signal handler and goes back to being pend. If the handler alters the execution path, via a call to **longjmp()** for example, and does not return then the task does not go back to being pend.

Signal handlers are typically defined as:

```
void sigHandler
(
int sig,                               /* signal number          */
)
{
    ...
}
```

In VxWorks, the signal handler is passed additional arguments and can be defined as:

```
void sigHandler
(
int sig,                               /* signal number          */
int code,                             /* additional code         */
struct sigcontext *pSigContext        /* context of task before signal */
)
{
    ...
}
```

The parameter *code* is valid only for signals caused by hardware exceptions. In this case, it is used to distinguish signal variants. For example, both numeric overflow and zero divide raise SIGFPE (floating-point exception) but have different values for *code*. (Note that when the above VxWorks extensions are used, the compiler may issue warnings.)

SIGNAL HANDLER DEFINITION

Signal handling routines must follow one of two specific formats, so that they may be correctly called by the operating system when a signal occurs.

Traditional signal handlers receive the signal number as the sole input parameter. However, certain signals generated by routines which make up the POSIX Real-Time Extensions (P1003.1b) support the passing of an additional application-specific value to the handler routine. These include signals generated by the **sigqueue()** call, by asynchronous I/O, by POSIX real-time timers, and by POSIX message queues.

If a signal handler routine is to receive these additional parameters, **SA_SIGINFO** must be set in the *sa_flags* field of the *sigaction* structure which is a parameter to the **sigaction()** routine. Such routines must take the following form:

```
void sigHandler (int sigNum, siginfo_t * pInfo, void * pContext);
```

Traditional signal handling routines must not set **SA_SIGINFO** in the *sa_flags* field, and must take the form of:

```
void sigHandler (int sigNum);
```

EXCEPTION PROCESSING

Certain signals, defined below, are raised automatically when hardware exceptions are encountered. This mechanism allows user-defined exception handlers to be installed. This is useful for recovering from catastrophic events such as bus or arithmetic errors. Typically, **setjmp()** is called to define the point in the program where control will be restored, and **longjmp()** is called in the signal handler to restore that context. Note that **longjmp()** restores the state of the task's signal mask. If a user-defined handler is not installed or the installed handler returns for a signal raised by a hardware exception, then the task is suspended and a message is logged to the console.

The following is a list of hardware exceptions caught by VxWorks and delivered to the offending task. The user may include the higher-level header file **sigCodes.h** in order to access the appropriate architecture-specific header file containing the code value.

If the configuration parameter **POSIX_SIGNAL_MODE** is defined as true VxWorks will generate signals on exception as defined by the POSIX 1003.1 specification. This mode will be automatically specified if the component **INCLUDE_RTP_POSIX_PSE52** is included.

If the **POSIX_SIGNAL_MODE** parameter is defined as false a backward compatible mode will be used. This mode will generate the signals used in previous versions of VxWorks when an exception occurs.

ARM

Signal Mode

VxWorks	POSIX	Code	Exception
SIGILL	SIGILL	EXC_OFF_RESET	branch through zero
SIGILL	SIGILL	EXC_OFF_UNDEF	undefined instruction
SIGILL	SIGILL	EXC_OFF_SWI	software interrupt
SIGSEGV	SIGSEGV	EXC_OFF_PREFETCH	instruction prefetch abort
SIGSEGV	SIGSEGV	EXC_OFF_DATA	data access

Coldfire

Signal Mode

VxWorks	POSIX	Code	Exception
SIGSEGV	SIGSEGV	IV_ACCESS_FAULT	access fault
SIGBUS	SIGBUS	IV_ADDRESS_ERROR	address error
SIGILL	SIGILL	IV_ILLEGAL_INSTRUCTION	illegal instruction
SIGFPE	SIGFPE	IV_ZERO_DIVIDE	divide by zero
SIGILL	SIGILL	IV_PRIVILEGE_VIOLATION	privelege instr violation
SIGTRAP	SIGTRAP	IV_TRACE	trace trap
SIGEMT	SIGTRAP	IV_LINE_1010_EMULATOR	line 1010 emulation
SIGEMT	SIGTRAP	IV_LINE_1111_EMULATOR	line 1111 emulation
SIGILL	SIGILL	IV_CP_PROTOCOL_VIOLATION	coprocessor protocol error
SIGFMT	SIGILL	IV_FORMAT_ERROR	exception frame format error

SIGFPE	SIGFPE	IV_FPCP_B_S_U_CONDITION	branch/set on unordered
SIGFPE	SIGFPE	IV_FPCP_INEXACT_RESULT	inexact result
SIGFPE	SIGFPE	IV_FP_UNDERFLOW	underflow
SIGFPE	SIGFPE	IV_FP_UNDERFLOW	overflow
SIGFPE	SIGFPE	IV_FP_OPERAND_ERROR	operand error
SIGFPE	SIGFPE	IV_SIGNALING_NAN	signaling NAN
SIGILL	SIGILL	IV_UNIMP_DATA_TYPE	unimplemented data type
SIGILL	SIGILL	IV_PMMU_CONFIGURATION	invalid MMU configuration
SIGILL	SIGILL	IV_PMMU_ILLEGAL_OPERATION	invalid MMU operation
SIGSEGV	SIGSEGV	IV_PMMU_ACCESS_LEVEL _VIOLATON	protected address access
SIGBUS	SIGBUS	IV_UNIMP_EFFECTIVE_ADDRESS	unimplemented effective addr
SIGILL	SIGILL	IV_UNIMP_INTEGER _INSTRUCTION	unimplemented integer instr

MIPS R3000/R4000

Signal Mode			
VxWorks	POSIX	Code	Exception
SIGBUS	SIGSEGV	BUS_TLBMOD	TLB modified
SIGBUS	SIGSEGV	BUS_TLBL	TLB miss on a load instruction
SIGBUS	SIGSEGV	BUS_TLBS	TLB miss on a store instruction
SIGBUS	SIGBUS	BUS_ADEL	address alignment error on load instr
SIGBUS	SIGBUS	BUS_ADES	address alignment error on store instr
SIGSEGV	SIGBUS	SEGV_IBUS	bus error (instruction)
SIGSEGV	SIGBUS	SEGV_DBUS	bus error (data)
SIGTRAP	SIGTRAP	TRAP_SYSCALL	syscall instruction executed
SIGTRAP	SIGTRAP	TRAP_BP	break instruction executed
SIGILL	SIGILL	ILL_ILLINSTR_FAULT	reserved instruction
SIGILL	SIGILL	ILL_COPROC_UNUSABLE	coprocessor unusable
SIGFPE	SIGFPE	FPE_FPA_UIO, SIGFPE	unimplemented FPA operation
SIGFPE	SIGFPE	FPE_FLTNAN_TRAP	invalid FPA operation
SIGFPE	SIGFPE	FPE_FLTDIV_TRAP	FPA divide by zero
SIGFPE	SIGFPE	FPE_FLTOVF_TRAP	FPA overflow exception
SIGFPE	SIGFPE	FPE_FLTUND_TRAP	FPA underflow exception
SIGFPE	SIGFPE	FPE_FLTINEX_TRAP	FPA inexact operation

Intel i386/i486

Signal Mode			
VxWorks	POSIX	Code	Exception

SIGILL	SIGILL	ILL_DIVIDE_ERROR	divide error
SIGEMT	SIGTRAP	EMT_DEBUG	debugger call.
SIGILL	SIGILL	ILL_NON_MASKABLE	NMI interrupt
SIGEMT	SIGTRAP	EMT_BREAKPOINT	breakpoint
SIGILL	SIGILL	ILL_OVERFLOW	INTO-detected overflow
SIGILL	SIGILL	ILL_BOUND	bound range exceeded
SIGILL	SIGILL	ILL_INVALID_OPCODE	invalid opcode
SIGFPE	SIGFPE	FPE_NO_DEVICE	device not available
SIGILL	SIGILL	ILL_DOUBLE_FAULT	double fault
SIGFPE	SIGFPE	FPE_CP_OVERRUN	coprocessor segment overrun
SIGILL	SIGILL	ILL_INVALID_TSS	invalid task state segment
SIGBUS	SIGBUS	BUS_NO_SEGMENT	segment not present
SIGBUS	SIGBUS	BUS_STACK_FAULT	stack exception
SIGILL	SIGILL	ILL_PROTECTION_FAULT	general protection
SIGBUS	SIGSEGV	BUS_PAGE_FAULT	page fault
SIGILL	SIGILL	ILL_RESERVED	(intel reserved)
SIGFPE	SIGFPE	FPE_CP_ERROR	coprocessor error
SIGBUS	SIGBUS	BUS_ALIGNMENT	alignment check

PowerPC

Signal Mode

VxWorks	POSIX	Code	Exception
SIGBUS	SIGBUS	_EXC_OFF_MACH	machine check
SIGBUS	SIGSEGV	_EXC_OFF_INST	instruction access
SIGBUS	SIGBUS	_EXC_OFF_ALIGN	alignment
SIGILL	SIGILL	_EXC_OFF_PROG	program
SIGBUS	SIGSEGV	_EXC_OFF_DATA	data access
SIGBUS	SIGSEGV	_EXC_OFF_PROT	data access (PPC405)
SIGFPE	SIGFPE	_EXC_OFF_FPU	floating point unavailable
SIGTRAP	SIGTRAP	_EXC_OFF_DBG	debug exception
SIGTRAP	SIGTRAP	_EXC_OFF_INST_BRK	inst. breakpoint
SIGTRAP	SIGTRAP	_EXC_OFF_TRACE	trace
SIGBUS	SIGBUS	_EXC_OFF_CRTL	critical interrupt
SIGILL	SIGILL	_EXC_OFF_SYSCALL	system call

Hitachi SH770x

Signal Mode

VxWorks	POSIX	Code	Exception
SIGSEGV	SIGSEGV	TLB_LOAD_MISS	TLB miss/invalid (load)
SIGSEGV	SIGSEGV	TLB_STORE_MISS	TLB miss/invalid (store)
SIGSEGV	SIGSEGV	TLB_INITIAL_PAGE_WRITE	Initial page write
SIGSEGV	SIGSEGV	TLB_LOAD_PROTEC_VIOLATION	TLB prot. violation (load)
SIGSEGV	SIGSEGV	TLB_STORE_PROTEC_VIOLATION	TLB prot. violation (store)
SIGBUS	SIGSEGV	BUS_LOAD_ADDRESS_ERROR	Address error (load)
SIGBUS	SIGSEGV	BUS_STORE_ADDRESS_ERROR	Address error (store)

SIGILL	SIGILL	ILLEGAL_INSTR_GENERAL	general illegal instruction
SIGILL	SIGILL	ILLEGAL_SLOT_INSTR	slot illegal instruction
SIGFPE	SIGILL	FPE_INTDIV_TRAP	integer zero divide

Hitachi SH7604/SH704x/SH703x/SH702x

Signal Mode

VxWorks	POSIX	Code	Exception
SIGILL	SIGILL	ILL_ILLINSTR_GENERAL	general illegal instruction
SIGILL	SIGILL	ILL_ILLINSTR_SLOT	slot illegal instruction
SIGBUS	SIGSEGV	BUS_ADDERR_CPU	CPU address error
SIGBUS	SIGSEGV	BUS_ADDERR_DMA	DMA address error
SIGFPE	SIGFPE	FPE_INTDIV_TRAP	integer zero divide

SIMNT simulator

Signal Mode

VxWorks	POSIX	Code	Exception
SIGFPE	SIGFPE	EXC_INT_DIVIDE_BY_ZERO	Integer zero divide
SIGEMT	SIGTRAP	EXC_SINGLE_STEP	Single step
SIGSEGV	SIGBUS	EXC_DATATYPE_MISALIGNMENT	Data alignment
SIGEMT	SIGTRAP	EXC_BREAKPOINT	Breakpoint
SIGSEGV	SIGSEGV	EXC_IN_PAGE_ERROR	In page error
SIGILL	SIGILL	EXC_ILLEGAL_INSTRUCTION	Illegal instruction
SIGILL	SIGILL	EXC_INVALID_DISPOSITION	Invalid disposition
SIGBUS	SIGSEGV	EXC_ARRAY_BOUNDS_EXCEEDED	Array bounds exceeded
SIGILL	SIGFPE	EXC_FLT_DENORMAL_OPERAND	Floating point denormalize
SIGFPE	SIGFPE	EXC_FLT_DIVIDE_BY_ZERO	Floating point zero divide
SIGFPE	SIGFPE	EXC_FLT_INEXACT_RESULT	Floating point inexact result
SIGFPE	SIGFPE	EXC_FLT_INVALID_OPERATION	Floating point invalid operation
SIGFPE	SIGFPE	EXC_FLT_OVERFLOW	Floating point overflow
SIGSEGV	SIGSEGV	EXC_ACCESS_VIOLATION	Access violation
SIGBUS	SIGBUS	EXC_FLT_STACK_CHECK	Floating point stack check
SIGFPE	SIGFPE	EXC_FLT_UNDERFLOW	Floating point underflow
SIGFPE	SIGFPE	EXC_INT_OVERFLOW	Integer overflow
SIGILL	SIGILL	EXC_PRIV_INSTRUCTION	Private instruction
SIGBUS	SIGBUS	EXC_STACK_OVERFLOW	Stack overflow
SIGILL	SIGILL	EXC_UNKNOWN	Unknown

SIMLINUX simulator

Signal Mode

VxWorks	POSIX	Code	Exception
SIGSEGV	SIGSEGV	IV_SEGV	Segmentation violation
SIGBUS	SIGBUS	IV_BUS	Bus error
SIGILL	SIGILL	IV_ILL	Illegal instruction
SIGFPE	SIGFPE	IV_FPE	Floating point exception
SIGTRAP	SIGTRAP	IV_TRAP	Trap

SIMSOLARIS simulator

Signal Mode	VxWorks	POSIX	Code	Exception
SIGSEGV	SIGSEGV	SIGSEGV	IV_SEGV	Segmentation violation
SIGBUS	SIGBUS	SIGBUS	IV_BUS	Bus error
SIGILL	SIGILL	SIGILL	IV_ILL	Illegal instruction
SIGFPE	SIGFPE	SIGFPE	IV_FPE	Floating point exception
SIGTRAP	SIGTRAP	SIGTRAP	IV_TRAP	Trap

Two signals are provided for application use: SIGUSR1 and SIGUSR2. VxWorks will never use these signals; however, other signals may be used by VxWorks in the future.

INCLUDE FILES **signal.h**

SEE ALSO **intLib**, *IEEE POSIX 1003.1b*, the VxWorks programmer guides.

smMemLib

NAME **smMemLib** – shared memory management library (VxMP Option)

ROUTINES **memPartSmCreate()** – create a shared memory partition (VxMP Option)
smMemAddToPool() – add memory to shared memory system partition (VxMP Option)
smMemOptionsSet() – set debug options for shared memory system partition (VxMP Option)
smMemMalloc() – allocate block of memory from shared memory system partition (VxMP Option)
smMemCalloc() – allocate memory for array from shared memory system partition (VxMP Option)
smMemRealloc() – reallocate block of memory from shared memory system partition (VxMP Option)
smMemFree() – free a shared memory system partition block of memory (VxMP Option)
smMemFindMax() – find largest free block in shared memory system partition (VxMP Option)

DESCRIPTION This library provides facilities for managing the allocation of blocks of shared memory from ranges of memory called shared memory partitions. The routine **memPartSmCreate()** is used to create shared memory partitions in the shared memory pool. The created partition can be manipulated using the generic memory partition calls, **memPartAlloc()**, **memPartFree()**, etc. (for a complete list of these routines, see the manual entry for **memPartLib**). The maximum number of partitions that can be created is determined by the configuration parameter **SM_OBJ_MAX_MEM_PART**.

The **smMem...()** routines provide an easy-to-use interface to the shared memory system partition. The shared memory system partition is created when the shared memory object facility is initialized.

Shared memory management information and statistics display routines are provided by **smMemShow**.

The allocation of memory, using **memPartAlloc()** in the general case and **smMemMalloc()** for the shared memory system partition, is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when freed using **memPartFree()** and **smMemFree()**.

There is a 28-byte overhead per allocated block (architecture dependent), and allocated blocks are aligned on a 16-byte boundary.

All memory used by the shared memory facility must be in the same address space, that is, it must be reachable from all the CPUs with the same offset as the one used for the shared memory anchor.

CONFIGURATION Before routines in this library can be called, the shared memory objects facility must be initialized by a call to **usrSmObjInit()**, which is found in **target/config/comps/src/usrSmObj.c**. This is done automatically by VxWorks when the **INCLUDE_SM_OBJ** component is included.

Various debug options can be selected for each partition using **memPartOptionsSet()** and **smMemOptionsSet()**. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. In both cases, options can be selected for system actions to take place when the error is detected: (1) return the error status, (2) log an error message and return the error status, or (3) log an error message and suspend the calling task.

One of the following options can be specified to determine the action to be taken when there is an attempt to allocate more memory than is available in the partition:

MEM_ALLOC_ERROR_RETURN

just return the error status to the calling task.

MEM_ALLOC_ERROR_LOG_MSG

log an error message and return the status to the calling task.

MEM_ALLOC_ERROR_LOG_AND_SUSPEND

log an error message and suspend the calling task.

The following option is specified by default to check every block freed to the partition. If this option is specified, **memPartFree()** and **smMemFree()** will make a consistency check of various pointers and values in the header of the block being freed.

MEM_BLOCK_CHECK

check each block freed.

One of the following options can be specified to determine the action to be taken when a bad block is detected when freed. These options apply only if the **MEM_BLOCK_CHECK** option is selected.

MEM_BLOCK_ERROR_RETURN

just return the status to the calling task.

MEM_BLOCK_ERROR_LOG_MSG

log an error message and return the status to the calling task.

MEM_BLOCK_ERROR_LOG_AND_SUSPEND

log an error message and suspend the calling task.

The default options when a shared partition is created are

MEM_ALLOC_ERROR_LOG_MSG, **MEM_BLOCK_CHECK**, **MEM_BLOCK_ERROR_RETURN**.

When setting options for a partition with **memPartOptionsSet()** or **smMemOptionsSet()**, use the logical OR operator between each specified option to construct the *options* parameter. For example:

```
memPartOptionsSet (myPartId, MEM_ALLOC_ERROR_LOG_MSG |
                    MEM_BLOCK_CHECK |
                    MEM_BLOCK_ERROR_LOG_MSG);
```

AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smMemLib.h
SEE ALSO	smMemShow , memLib , memPartLib , smObjLib , usrSmObjInit() , the VxWorks programmer guides.

smMemShow

NAME	smMemShow – shared memory management show routines (VxMP Option)
ROUTINES	smMemShow() – show the shared memory system partition blocks and statistics (VxMP Option)
DESCRIPTION	This library provides routines to show the statistics on a shared memory system partition.

General shared memory management routines are provided by **smMemLib**.

CONFIGURATION	The routines in this library are included by default if the component INCLUDE_SM_OBJ is included.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smLib.h , smObjLib.h , smMemLib.h
SEE ALSO	smMemLib , the VxWorks programmer guides.

smNameLib

NAME	smNameLib – shared memory objects name database library (VxMP Option)
ROUTINES	smNameAdd() – add a name to the shared memory name database (VxMP Option) smNameFind() – look up a shared memory object by name (VxMP Option) smNameFindByValue() – look up a shared memory object by value (VxMP Option) smNameRemove() – remove an object from the shared memory objects name database (VxMP Option)
DESCRIPTION	<p>This library provides facilities for managing the shared memory objects name database. The shared memory objects name database associates a name and object type with a value and makes that information available to all CPUs. A name is an arbitrary, null-terminated string. An object type is a small integer, and its value is a global (shared) ID or a global shared memory address.</p> <p>Names are added to the shared memory name database with smNameAdd(). They are removed by smNameRemove().</p> <p>Objects in the database can be accessed by either name or value. The routine smNameFind() searches the shared memory name database for an object of a specified name. The routine smNameFindByValue() searches the shared memory name database for an object of a specified identifier or address.</p> <p>Name database contents can be viewed using smNameShow().</p> <p>The maximum number of names to be entered in the database is defined in the configuration parameter SM_OBJ_MAX_NAME. This value is used to determine the size of a dedicated shared memory partition from which name database fields are allocated.</p> <p>The estimated memory size required for the name database can be calculated as follows:</p>

$$\text{name database pool size} = \text{SM_OBJ_MAX_NAME} * 40 \text{ (bytes)}$$

The display facility for the shared memory objects name database is provided by the **smNameShow** module.

EXAMPLE

The following code fragment allows a task on one CPU to enter the name, associated ID, and type of a created shared semaphore into the name database. Note that CPU numbers can belong to any CPU using the shared memory objects facility.

On CPU 1:

```
#include "vxWorks.h"
#include "semLib.h"
#include "smNameLib.h"
#include "semSmLib.h"
#include "stdio.h"

testSmSem1 (void)
{
    SEM_ID smSemId;

    /* create a shared semaphore */

    if ((smSemId = semBSmCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL)
    {
        printf ("Shared semaphore creation error.");
        return (ERROR);
    }

    /*
     * make created semaphore Id available to all CPUs in
     * the system by entering its name in shared name database.
     */

    if (smNameAdd ("smSem", smSemId, T_SM_SEM_B) != OK )
    {
        printf ("Cannot add smSem into shared database.");
        return (ERROR);
    }
    ...

    /* now use the semaphore */

    semGive (smSemId);
    ...
}
```

On CPU 2:

```
#include "vxWorks.h"
#include "semLib.h"
#include "smNameLib.h"
#include "stdio.h"

testSmSem2 (void)
{
    SEM_ID smSemId;
```

```
int    objType;    /* place holder for smNameFind() object type */

/* get semaphore ID from name database */

smNameFind ("smSem", (void **) &smSemId, &objType, WAIT_FOREVER);
    ...
/* now that we have the shared semaphore ID, take it */

semTake (smSemId, WAIT_FOREVER);
    ...
}
```

CONFIGURATION	Before routines in this library can be called, the shared memory object facility must be initialized by calling usrSmObjInit() . This is done automatically during VxWorks initialization when the component INCLUDE_SM_OBJ is included.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smNameLib.h
SEE ALSO	smNameShow , smObjLib , smObjShow , usrSmObjInit() , the VxWorks programmer guides.

smNameShow

NAME	smNameShow – shared memory objects name database show routines (VxMP Option)
ROUTINES	smNameShow() – show the contents of the shared memory objects name database (VxMP Option)
DESCRIPTION	This library provides a routine to show the contents of the shared memory objects name database. The shared memory objects name database facility is provided by the smNameLib module.
CONFIGURATION	The routines in this library are included by default if the component INCLUDE_SM_OBJ is included.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smNameLib.h
SEE ALSO	smNameLib , smObjLib , the VxWorks programmer guides.

smObjLib

NAME	smObjLib – shared memory objects library (VxMP Option)
ROUTINES	smObjLibInit() – install the shared memory objects facility (VxMP Option) smObjSetup() – initialize the shared memory objects facility (VxMP Option) smObjInit() – initialize a shared memory objects descriptor (VxMP Option) smObjAttach() – attach the calling CPU to the shared memory objects facility (VxMP Option) smObjLocalToGlobal() – convert a local address to a global address (VxMP Option) smObjGlobalToLocal() – convert a global address to a local address (VxMP Option) smObjTimeoutLogEnable() – control logging of failed attempts to take a spin-lock (VxMP Option)
DESCRIPTION	<p>This library contains miscellaneous functions used by the shared memory objects facility (VxMP). Shared memory objects provide high-speed synchronization and communication among tasks running on separate CPUs that have access to a common shared memory. Shared memory objects are system objects (e.g., semaphores and message queues) that can be used across processors.</p> <p>The main uses of shared memory objects are interprocessor synchronization, mutual exclusion on multiprocessor shared data structures, and high-speed data exchange.</p> <p>Routines for displaying shared memory objects statistics are provided by the smObjShow module.</p>

SHARED MEMORY MASTER CPU

One CPU node acts as the shared memory objects master. This CPU initializes the shared memory area and sets up the shared memory anchor. These steps are performed by the master calling **smObjSetup()**. This routine should be called only once by the master CPU. Usually **smObjSetup()** is called from **usrSmObjInit()**. (See "Configuration" below.)

Once **smObjSetup()** has completed successfully, there is little functional difference between the master CPU and other CPUs using shared memory objects, except that the master is responsible for maintaining the heartbeat in the shared memory objects header.

ATTACHING TO SHARED MEMORY

Each CPU, master or non-master, that will use shared memory objects must attach itself to the shared memory objects facility, which must already be initialized.

Before it can attach to a shared memory region, each CPU must allocate and initialize a shared memory descriptor (**SM_DESC**), which describes the individual CPU's attachment to the shared memory objects facility. Since the shared memory descriptor is used only by the local CPU, it is not necessary for the descriptor itself to be located in shared memory. In fact, it is preferable for the descriptor to be allocated from the CPU's local memory, since local memory is usually more efficiently accessed.

The shared memory descriptor is initialized by calling **smObjInit()**. This routine takes a number of parameters which specify the characteristics of the calling CPU and its access to shared memory.

Once the shared memory descriptor has been initialized, the CPU can attach itself to the shared memory region. This is done by calling **smObjAttach()**.

When **smObjAttach()** is called, it verifies that the shared memory anchor contains the value **SM_READY** and that the heartbeat located in the shared memory objects header is incrementing. If either of these conditions is not met, the routine will check periodically until either **SM_READY** or an incrementing heartbeat is recognized or a time limit is reached. The limit is expressed in seconds, and 600 seconds (10 minutes) is the default. If the time limit is reached before **SM_READY** or a heartbeat is found, **ERROR** is returned and **errno** is set to **S_smLib_DOWN**.

ADDRESS CONVERSION

This library also provides routines for converting between local and global shared memory addresses, **smObjLocalToGlobal()** and **smObjGlobalToLocal()**. A local shared memory address is the address required by the local CPU to reach a location in shared memory. A global shared memory address is a value common to all CPUs in the system used to reference a shared memory location. A global shared memory address is always an offset from the shared memory anchor.

SPIN-LOCK MECHANISM

The shared memory objects facilities use a spin-lock mechanism based on an indivisible read-modify-write (RMW) operation on a shared memory location which acts as a low-level mutual exclusion device. The spin-lock mechanism is called with a system-wide configuration parameter, **SM_OBJ_MAX_TRIES**, which specifies the maximum number of RMW tries on a spin-lock location.

Care must be taken that the number of RMW tries on a spin-lock on a particular CPU never reaches **SM_OBJ_MAX_TRIES**, otherwise system behavior becomes unpredictable. The default value should be sufficient for reliable operation.

The routine **smObjTimeoutLogEnable()** can be used to enable or disable the printing of a message should a shared memory object call fail while trying to take a spin-lock.

RELATION TO BACKPLANE DRIVER

Shared memory objects and the shared memory network (backplane) driver use common underlying shared memory utilities. They also use the same anchor, the same shared memory header, and the same interrupt when they are used at the same time.

LIMITATIONS

A maximum of twenty CPUs can be used concurrently with shared memory objects. Each CPU in the system must have a hardware test-and-set (TAS) mechanism, which is called via the system-dependent routine **sysBusTas()**.

The use of shared memory objects raises interrupt latency, because internal mechanisms lock interrupts while manipulating critical shared data structures. Interrupt latency does not depend on the number of objects or CPUs used.

GETTING STATUS INFORMATION

The routine **smObjShow()** displays useful information regarding the current status of shared memory objects, including the number of tasks using shared objects, shared semaphores, and shared message queues, the number of names in the database, and also the maximum number of tries to get spin-lock access for the calling CPU.

CONFIGURATION	When the component INCLUDE_SM_OBJ is included, the init and setup routines in this library are called automatically during VxWorks initialization.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smObjLib.h
SEE ALSO	smObjShow , semSmLib , msgQSmLib , smMemLib , smNameLib , usrSmObjInit() , the VxWorks programmer guides.

smObjShow

NAME	smObjShow – shared memory objects show routines (VxMP Option)
ROUTINES	smObjShow() – display the current status of shared memory objects (VxMP Option)
DESCRIPTION	This library provides routines to show shared memory object statistics, such as the current number of shared tasks, semaphores, message queues, etc.
CONFIGURATION	The routines in this library are included by default if the component INCLUDE_SM_OBJ is included.
AVAILABILITY	This module is distributed as a component of the unbundled shared memory objects support option, VxMP.
INCLUDE FILES	smObjLib.h
SEE ALSO	smObjLib , the VxWorks programmer guides.

smpLockDemo

NAME	smpLockDemo – synchronization mechanism demo for VxWorks SMP
ROUTINES	smpLockDemo() – smpLockDemo entry point (shell command)
DESCRIPTION	<p>This module contains code to demonstrate the capabilities of VxWorks SMP to provide CPU synchronization mechanisms. In an SMP system tasks or ISRs running concurrently on different CPU must coordinate their access to shared data. For that purpose VxWorks SMP offers a number of mechanisms:</p> <ul style="list-style-type: none">- Semaphores: These can be used for synchronization between tasks or when an ISR wants to "wake-up" as task.- VxWorks Events: These can be used for synchronization between tasks or when an ISR wants to "wake-up" as task.- Atomic Operators: These can be used for the purpose of safely performing a simple read-modify-write of memory from either tasks or ISRs. For example for incrementing a global count of some sort.- Spinlocks: These can be used for synchronization between tasks, between ISRs or between tasks and ISRs. This synchronization is typically required when non-trivial shared data manipulation is required or for protecting a critical section of code.

This demo shows comparative performance figures for atomic operators, spinlocks and highlights the fact it is absolutely critical to use a synchronization mechanism to preserve the integrity of shared data when it is manipulated by more than one task on an SMP system.

DEMO IMPLEMENTATION

The smpLockDemo consists of two equal priority tasks, each trying to repeatedly (loop) increment a shared global count for a user-defined amount of time. Each task also has a local count that it is incremented every time it increments the global count. Assuming no corruption takes place the sum of the local counts, which are safely incremented without the need for synchronization, should be equal to the global count.

The procedure above is repeated five times, using a different synchronization method:

- **No synchronization:** The tasks do not synchronize their accesses to the global count.
- **Spinlocks:** The tasks acquire a spinlock, increment the global count and then release the spinlock.
- **vxAtomicInc() Operator:** The tasks use the **vxAtomicInc()** operator to atomically increment the global count.
- **vxTas() Operator:** The tasks atomically set or clear a flag using **vxTas()** and use that flag as a custom semaphore. When the flag is cleared it means the semaphore is not

available. When it is set it means the semaphore is available. The tasks then acquire the semaphore, increment the global count and release the semaphore.

- **vxAtomicAdd()** Operator: The tasks use the **vxAtomicAdd()** operator to atomically increment the global count.

The demo then displays the results as described below.

INVOKING THE DEMO

The demo can be invoked from the kernel shell:

```
-> smpLockDemo 3
```

The optional argument is the number of seconds the demo spends updating the local and global counts for each synchronization mechanism mentioned above. In this case, specifying "3" means the demo will run for approximately 15 seconds (3 sec x 5 sync mechanisms).

If no argument is supplied the default is to use a two-second time period.

INTERPRETING DEMO RESULTS

The demo displays results similar to the ones below on standard output.

```
-> smpLockDemo
```

METHOD	TASK 0	TASK 1	SUM(COUNTS)	GLOBAL	RESULT
no-lock	0x253d8ae	0x253b31c	0x4a78bca	0x470327d	Failed
spinLocks	0x782f43	0x78265a	0xf0559d	0xf0559d	Passed
atomicInc	0x20b600d	0x20b4623	0x416a630	0x416a630	Passed
test-and-set	0x1509e72	0x150a628	0x2a1449a	0x2a1449a	Passed
atomic Add	0x1e25b2a	0x1e26d2a	0x3c4c854	0x3c4c854	Passed

The METHOD column indicates the type of synchronization used by the tasks to safely update the global count. The TASK0 column displays the local count for task 0. The figure indicates the number of times the task was able to execute its loop within the user-defined time period. The greater the number the better the performance of the synchronization mechanism. The TASK1 column displays the local count for task 1. The SUM(COUNTS) column is the sum of the previous two columns. The GLOBAL column is the value of the global count; that is, the total number of times the tasks were able to execute their loop. The RESULT column simply indicates whether or not the sum of the local counts is equal to that of the global count.

As can be seen from the example above, not using a synchronization method to update the global count causes severe corruption, even though incrementing a variable only offers a small window of opportunity for corruption to take place. The results also show that using the **vxAtomicInc()** operator is safe and faster than using **vxAtomicAdd()**, which in turn is faster than using a spinlock. The reason atomic operators are faster than spinlocks is because they are designed for simple atomic read-modify-write operations, which is exactly what this demo is. Spinlocks are meant for the synchronization of more complex

operations. The test-and-set approach uses **vxTas()** to implement a custom semaphore. This method is slower than atomic operators but faster than spinlocks, It is however subject to live lock and hence should be used with extreme caution. Spinlocks are a much safer alternative to the custom semaphore.

Below is an example using the same time period and the same platform but only with one CPU enabled. As can be seen corruption did not take place since the two tasks did not run concurrently. Furthermore, the counts indicate a better performance than the example above. This is because there is much less contention for the global count since tasks do not run concurrently. In some way this is an example of an application that does not benefit from concurrent execution because of the excessive contention.

-> smpLockDemo

METHOD	TASK 0	TASK 1	SUM(COUNTS)	GLOBAL	RESULT
-----	-----	-----	-----	-----	-----
no-lock	0x265d794	0x2675ee8	0x4cd367c	0x4cd367c	Passed
spinLocks	0xaae678	0xaaefaf	0x155d627	0x155d627	Passed
atomicInc	0x23c5135	0x23c7018	0x478c14d	0x478c14d	Passed
test-and-set	0x1a84dd6	0x1a864c0	0x350b296	0x350b296	Passed
atomic Add	0x20b9c80	0x20bb843	0x41754c3	0x41754c3	Passed

INCLUDE FILES none

SEE ALSO spinLockLib, vxAtomicLib, smpLockDemo()

snsLib

NAME snsLib – Socket Name Service library

ROUTINES

DESCRIPTION This library implements the Socket Name Service (SNS). SNS allows applications based on the Socket Application Library (SAL) to associate a list of socket addresses with a service name. This name can then be referenced by other SAL-based applications to determine which socket addresses the server application providing the specified service is using.

SERVICE INFORMATION

SNS maintains a simple database of service entries. Each service entry contains the following information:

Service Name:

A character string mnemonic for the service.

Service Scope:

Level of visibility of the service within the system.

Service Sockets:

Information about the sockets which provide the service.

Service Owner:

The entity that created the service (operating system kernel or RTP identifier).

SERVICE LOCATOR

An application that wishes to register a new service, or locate an existing service, must specify a "service location". The service location is simply the service's name, optionally attached with a scope indicator in URL format. All locations must be unique within a scope.

SERVICE SCOPE

The service scoping capability of SNS allows a server application to limit the visibility of a service name to a specified subset of applications within a system. An analogous capability allows a client application searching for a specified service name to limit how far SNS should search. Thus, a search only returns a matching entry if the search scope specified by the client overlaps the service scope specified by the server. Four levels of scope are supported:

private:

The service is visible within the service's memory region (the operating system kernel space or RTP) only.

node:

The service is visible within the service's owner local node only.

cluster:

The service is visible within the service's cluster of nodes only.

system:

The service is visible to all nodes in the system.

The scoping capability of SNS is best illustrated by visualizing the SNS name space as a set of nested boxes, each representing a different scope.

SNS currently supports the exchange of service information between nodes using the TIPC (Transparent Inter-Process Communication) protocol. Thus the TIPC component must be included in a project to utilize the distributed mode of operation. Services with a scope of the "system" and "cluster" will be visible to an application on another node (if the address family allows it).

It is possible to create **AF_LOCAL** sockets with scope larger than the node, but these sockets will not be visible outside of the node on which they were created.

URL SCHEME SYNOPSIS

[SNS:]service_name[@scope]

where the parts in brackets, [], are optional.

SNS: represent the URL service scheme, i.e. the Socket Name Service. It is the only scheme accepted and can be omitted.

@scope represents the visibility of the service name within the system. It can take several values, depending from the context and the application needs. If the the scope is not specified, "@node" is assumed.

The URL representation is case insensitive.

For SAL service creation, registration or removal **service_name** cannot contain any wildcard symbol, and scope must be the exact scope such as **node**, **private**, **cluster** and **system**. **service_name** should not contain RFC 2396 reserved characters.

For the SAL client (to open or find services), **service_name** make contain wildcard, and scope may provide exact scope or the outmost scope. For detail, refer to the **SERVICE DISCOVERY** section below.

SERVICE REGISTRATION AND DISCOVERY

A service who wants to take advantage of the SNS capability, registers itself to the system by providing an URL format identifier. If no scope is specified, the default is set to **node**.

A client discovers a server by specifying the service URL. In this case, there are two search options for each level of visibility:

- if a user specifies the service URL with the scope as described above, the system looks for the service only within the specified scope.
- if the scope is prefixed with **upto_**, such as **upto_node**, the system searches a service beginning from the **private** scope. If it cannot find one, the search moves outward to the next scope. The search stops either when a service is located or the specified scope has been reached and no service was found.

For example, assuming both client and server are in the same node, if a service is defined with **node** scope and the client specifies a scope **upto_cluster** the search will return the matching service. On the other hand, if the client specifies **cluster** then the search will not return that service. It might still return another service with the same name but in a different node, which registered itself with **cluster** visibility.

CONFIGURATION Socket Name Service capabilities are provided by an SNS server task, which can be configured to start automatically when VxWorks starts up. The server task can be configured to run in its own RTP or as part of the base operating system.

To use the SNS server, configure VxWorks with either the **INCLUDE_SNS** or the **INCLUDE_SNS_RTP** component. With either component you will also require **INCLUDE_UN_COMP**, **INCLUDE_SAL_COMMON**, and **INCLUDE_SAL_SERVER**.

For the distributed versions of the SNS server, the respective components are **INCLUDE_SNS_MP** and **INCLUDE_SNS_MP_RTP**. Note that an additional task called **dsalMonitor** is started in the kernel to monitor all existing distributed SNS servers in the system.

If the SNS server runs as an RTP, the executable needs to be allocated in the path defined by **SNS_PATHNAME**.

INCLUDE FILES **snsLib.h**

SEE ALSO **salClient, salServer, snsShow**

snsShow

NAME **snsShow** – Socket Name Service show routines

ROUTINES **snsShow()** – show information about services in the SNS directory

DESCRIPTION This library provides routines to show information about the services registered with Socket Name Service (SNS), including the name and scope of the service and the socket address(es) of the server application that provides the service.

 An SNS server task must be operational, otherwise no information can be obtained about SNS.

CONFIGURATION To use the SNS show facility, configure VxWorks with the **INCLUDE_SNS_SHOW** component.

INCLUDE FILES **snsLib.h**

SEE ALSO **snsLib**

spinLockLib

NAME **spinLockLib** – spinlock library

ROUTINES **spinLockIsrInit()** – initialize an ISR-callable spinlock
spinLockIsrTake() – take an ISR-callable spinlock
spinLockIsrGive() – release an ISR-callable spinlock
spinLockTaskInit() – initialize a task-only spinlock
spinLockTaskTake() – take a task-only spinlock
spinLockTaskGive() – release a task-only spinlock
spinLockIsrHeld() – is an ISR-callable spinlock held by the current CPU?

DESCRIPTION Spinlocks are a fundamental methods of synchronization between CPUs in a multi-CPU system like VxWorks SMP. They are meant to control accesses to a shared resource or a short critical section of code. The **taskCpuLock()** and **intCpuLock()** routines provide

similar mutual exclusion capabilities in the uniprocessor version of VxWorks but they are not suitable in an SMP environment. However the same set of APIs is available for UP for ease of portability between SMP and UP applications.

In the information below, when more than one CPU is mentioned, the information is applicable only to SMP. Unless explicitly mentioned, the information below is relevant for both SMP and UP.

Conceptually spinlocks are similar to mutual exclusion semaphores but they differ in several ways:

- Because a spinlock is a CPU synchronization mechanism, the real owner of a spinlock is the CPU, not the task nor ISR that acquires the lock. Therefore the concept of pending while waiting for the lock does not apply as it does for a semaphore since a CPU cannot be pended. Instead of pending, a CPU busy-waits for a spinlock to become available. In UP a spinlock is always available for a CPU since no additional CPUs compete against it. Therefore the CPU never busy-waits in the UP environment.
- A spinlock cannot order its access according to the priority of the tasks seeking to acquire it. Hence priority inversions are inherent in the acquisition of a spinlock. However, given that a spinlock should only be held for a very short periods of time, and moreover held for a deterministic amount of time, any priority inversions should only occur for a deterministic amount of time.
- Unlike a semaphore, a spinlock can be acquired by an ISR. This allows synchronization between tasks and ISRs.

TYPES OF SPINLOCKS

Because a CPU really owns a spinlock, it is imperative that the thread of execution (task or ISR) that acquired the lock not be pre-empted until it releases the lock. Allowing pre-emption can cause live lock, which is a state where a CPU busy-waits for a spinlock that is never going to become available. This is why VxWorks SMP introduces two types of spinlocks, each type having its own purpose and having different pre-emption characteristics:

- ISR-callable spinlocks can be acquired by both tasks and ISRs. Due to the live lock scenario implied when an ISR goes to acquire a spinlock held by a task on the same CPU, interrupts must be disabled on that CPU when an ISR-callable spinlock is held by either a task or an ISR. Furthermore, a task that acquires an ISR-callable spinlock cannot be rescheduled.
- Task-only spinlocks can be taken by tasks only. Due to the live lock implied when a task goes to acquire a spinlock held by another task on the same CPU, rescheduling on that CPU must not take place. Interrupts are not disabled on the local CPU when a task-only spinlock is held, but pre-emption is disabled. Due to its busy-wait nature, task-only spinlocks should be used only if the length of time required for the spinlock is very short. Otherwise semaphores should be used for synchronization that requires longer duration.

Because of the impact on the ability to process interrupts and perform scheduling, it is essential that spinlocks only be acquired for very short durations; just like **taskCpuLock()** and **intCpuLock()** must only be used for short durations in a uniprocessor VxWorks system. Acquisition of a spinlock by a CPU does not have any effect on the ability of other CPUs to process interrupts or perform scheduling in a VxWorks SMP system.

A task or ISR must not take more than one spinlock at a time. This is to prevent live lock situations inherent with schemes that require the sequential acquisition and release of multiple spinlocks. Conceptually this is very similar to deadlock situations that can happen with semaphores. The major difference with spinlocks is that the acquiring entity never pends or blocks while waiting for a resource like it is the case for semaphores. Instead of deadlock the system goes into a live lock where a CPU continuously attempts to acquire a spinlock without success. Because either task pre-emption or interrupts are disabled on the CPU in question during this time, there is no way for the CPU to come out of that state safely. The symptoms will make the situation appear as if the CPU is "hung". If operating in a UP system, the acquisition of a spinlock always succeeds even if previously acquired. However acquired recursively will provoke an unstable behaviour.

SPINLOCKS AS MEMORY BARRIERS

In a multi-processor system it is typical that memory access are weakly ordered. For that reason both types of spinlock perform full memory barriers after acquiring a spinlock and before releasing it. The term full memory memory barrier implies that both READ and WRITE memory access are forced to be performed in order. Data structures updated while a CPU is holding a lock will be completed before a lock is released or acquired.

SPINLOCK RESTRICTED APIS

Internally the VxWorks kernel uses spinlocks to protect its critical sections. The scenarios under which this is done make it very likely that an application would cause the system to enter into a live lock state if a task/ISR were to enter a kernel critical section while it already has a spinlock. For this reason a number of the VxWorks kernel APIs are spinlock restricted meaning that calling the API while holding a spinlock is not permitted. The reference entries in the VxWorks Kernel API reference manual specifies when this restriction applies to an API.

DIFFERENCES BETWEEN SMP and UP SPINLOCKS

- Spinlock APIs have identical interfaces between SMP and UP but behave differently. In UP spinlocks do not perform a memory fencing operation nor explicit bus reservation while acquiring a lock. Therefore the spinlock APIs provided by this library can not be used to access atomically shared memory in an AMP system.
- In UP spinlocks will not cause a live lock since busy-wait operation is not performed, but taking a previously taken lock may cause fatal problems.
- In SMP mode, the spinlock library offer the capability to enable debugging in order to catch improper usages. This feature allows to enforce SMP restrictions on usages of the spinlock APIs. In case of error detection a kernel fatal ED&R error message is injected which may bring the whole system down based on the policy put in place. The

different scenarios that may cause a fatal kernel ED&R error is documented in the description of the concerned API. By default debugging is not enabled. One may enable debugging by including the following component while building a VxWorks image: **INCLUDE_SPINLOCK_DEBUG**

In order to disable spinlock debugging, simply include **INCLUDE_SPINLOCK** and **INCLUDE_SPINLOCK_DEBUG** is automatically removed by the configuration tool.

EXAMPLE

A spinlock controls access to a shared resource between two or more processors. The resource shared in this instance is a global variable that may be updated by two or more tasks that could be running simultaneously on separate processors. The spinlock will ensure that a global count is exclusively accessed by one core at a given time in order to increment it atomically.

If the global count is accessed from an ISR it is necessary to use an ISR-callable, otherwise a task-only spinlock is sufficient.

Initially the spinlock must be declared and initialized. If it is required to define the spinlock statically then declare it as follows:

SPIN_LOCK_XX_DECL (mySpinLock, 0); where XX is defined as either "ISR" or "TASK".

The same can be dynamically accomplished just before taking the lock as follows:

```
void foo (void)
{

    /* automatic variable declaration */

    spinLockxx_t mySpinLock; where xx is defined as "Isr" or "Task"

    SPIN_LOCK_XX_INIT (&mySpinLock, 0);

    /* Before accessing the shared ressource the spinlock must be acquired */

    SPIN_LOCK_XX_TAKE (&mySpinLock);

    /* ... Access the share resource here */

    /* Now release the lock */

    SPIN_LOCK_XX_GIVE (&mySpinLock);
}
```

INCLUDE FILES none

spyLib

NAME spyLib – spy CPU activity library

ROUTINES spyLibInit() – initialize task cpu utilization tool package

DESCRIPTION This library provides a facility to monitor tasks' use of CPUs. The primary interface routine, **spy()**, periodically calls **spyReport()** to display the amount of CPU time utilized by each task, the amount of time spent at interrupt level, the amount of time spent in the kernel, and the amount of idle time. It also displays the total usage since the start of **spy()** (or the last call to **spyClkStart()**), and the change in usage since the last **spyReport()**.

CPU usage can also be monitored manually by calling **spyClkStart()** and **spyReport()**, instead of **spy()**. In this case, **spyReport()** provides a one-time report of the same information provided by **spy()**.

Data is gathered by an interrupt-level routine that is connected by **spyClkStart()** to the auxiliary clock. Currently, this facility cannot be used with CPUs that have no auxiliary clock. Interrupts that are at a higher level than the auxiliary clock's interrupt level cannot be monitored.

All user interface routines except **spyLibInit()** are available through **usrLib**. Help information concerning spy can be obtained by calling the **spyHelp()** routine in the target shell.

EXAMPLE The following call:

```
-> spy 10, 100
```

will generate a report in the following format every 10 seconds, gathering data at the rate of 100 times per second.

NAME	ENTRY	TID	PRI	total % (ticks)	delta % (ticks)
tExcTask	0x600abbd0	0x601c7af8	0	0% (0)	0% (0)
tJobTask	0x600acce4	0x603c6010	0	0% (0)	0% (0)
tLogTask	logTask	0x603ca010	0	0% (0)	0% (0)
tNbioLog	0x600ae030	0x603ce010	0	0% (0)	0% (0)
tShell0	shellTask	0x60559fc0	1	0% (4)	0% (0)
tWdbTask	wdbTask	0x60545fa0	3	0% (0)	0% (0)
tSpyTask	spyComTask	0x60577020	5	0% (25)	0% (1)
tAioIoTask1	aioIoTask	0x6048fa60	50	0% (0)	0% (0)
tAioIoTask0	aioIoTask	0x6047f020	50	0% (0)	0% (0)
tNetTask	netTask	0x604b3020	50	0% (0)	0% (0)
tAioWait	aioWaitTask	0x604716d0	51	0% (0)	0% (0)
tTestTask	test	0x6170ef78	100	71% (4683)	99% (500)

KERNEL	0%	(0)	0%	(0)
INTERRUPT	0%	(0)	0%	(0)
IDLE	27%	(1812)	0%	(0)
TOTAL	98%	(6524)	99%	(501)

The "total" column reflects CPU activity since the initial call to **spy()** or the last call to **spyClkStart()**. The "delta" column reflects activity since the previous report. A call to **spyReport()** will produce a single report; however, the initial auxiliary clock interrupts and data collection must first be started using **spyClkStart()**.

Data collection/clock interrupts and periodic reporting are stopped by calling:

-> spyStop

SMP CONSIDERATIONS

When used on an SMP system, spy reports changes to the following :

NAME	ENTRY	TID	PRI	total % (ticks)	delta % (ticks)
tExcTask	0x600b72e4	0x601e21e0	0	0% (0)	0% (0)
tJobTask	0x600bf6b4	0x6042a010	0	0% (0)	0% (0)
tLogTask	logTask	0x6042e010	0	0% (0)	0% (0)
tNbioLog	0x600d3420	0x60432010	0	0% (0)	0% (0)
tShell0	shellTask	0x605c20c8	1	0% (10)	0% (10)
tWdbTask	wdbTask	0x605adfa0	3	0% (0)	0% (0)
tAioIoTask1	aioIoTask	0x604f7a60	50	0% (0)	0% (0)
tAioIoTask0	aioIoTask	0x604e7020	50	0% (0)	0% (0)
tNetTask	netTask	0x6051baf0	50	0% (0)	0% (0)
tAioWait	aioWaitTask	0x604d9768	51	0% (0)	0% (0)
worker	0x60165110	0x61716af0	100	7% (202)	7% (202)
worker	0x60165110	0x605df108	100	7% (202)	7% (202)
worker	0x60165110	0x605df690	100	3% (93)	3% (93)
worker	0x60165110	0x6171cf70	100	6% (193)	6% (193)
worker	0x60165110	0x6171d3d0	100	7% (202)	7% (202)
worker	0x60165110	0x6171d830	100	7% (200)	7% (200)
worker	0x60165110	0x6062b020	100	7% (201)	7% (201)
worker	0x60165110	0x6062b480	100	7% (200)	7% (200)
worker	0x60165110	0x6062b8e0	100	7% (202)	7% (202)
worker	0x60165110	0x60639020	100	7% (200)	7% (200)
tIdleTask0	idleTaskEntr	0x603be000	287	77% (2156)	77% (2156)
tIdleTask1	idleTaskEntr	0x603d6450	287	76% (2137)	76% (2137)
tIdleTask2	idleTaskEntr	0x603ee8a0	287	73% (2055)	73% (2055)
tIdleTask3	idleTaskEntr	0x60406cf0	287	100% (2799)	100% (2799)
KERNEL				5% (144)	5% (144)
INTERRUPT				0% (0)	0% (0)
TOTAL				396% (2799)	396% (2799)

CPU	KERNEL	INTERRUPT	IDLE	TASK	TOTAL
0	0% (11)	0% (0)	77% (2156)	22% (632)	99%
1	1% (42)	0% (0)	76% (2137)	22% (620)	99%
2	3% (91)	0% (0)	73% (2055)	23% (653)	99%
3	0% (0)	0% (0)	100% (2799)	0% (0)	100%

The first, uniprocessor-style report is kept, the main differences being :

- there is no IDLE total anymore, this state being shown via the idle tasks.
- since there is true concurrent execution, the TOTAL percentage will go beyond 100% ($n * 100\%$, where n is the number of CPUs in the system). The number of TOTAL ticks is however left unchanged as it shows the real duration of the measurement (2799 ticks in the abovementioned example).

The second report shows, for each configured CPU, the number of ticks spent in each state.

LIMITATIONS	<p>On SMP systems, if a task migrates between CPUs during measurement, it is possible that it ends up being counted several times.</p> <p>Due to rounding done by spyLib display routines, the TOTAL percentage may not always reach 100%.</p>
INCLUDE FILES	spyLib.h
SEE ALSO	usrLib

ssiDb

NAME	ssiDb – SSI database module
ROUTINES	ssmCompRegister() – Register a component with SSI Manager. ssmCompInfoGet() – Get component information. ssiDbInit() – Initialize SSI database. ssiShow() – Display SSI information
DESCRIPTION	<p>The SSI database module maintains the info data of components participating in the SSI process. The module provides APIs to add or remove components from the database. It also provides APIs to allow changing the dependency info of a given component.</p> <p>The module also provides an API to generate the SSI dependency tree, which does play an important role in sequencing the startup and initialization of components.</p>
INCLUDE FILES	ssm.h

strSearchLib

NAME	strSearchLib – Efficient string search library
------	---

ROUTINES	fastStrSearch() – Search by optimally choosing the search algorithm bmsStrSearch() – Search using the Boyer-Moore-Sunday (Quick Search) algorithm bfStrSearch() – Search using the Brute Force algorithm
DESCRIPTION	<p>This library supplies functions to efficiently find the first occurrence of a string (called a pattern) in a text buffer. Neither the pattern nor the text buffer needs to be null-terminated.</p> <p>The functions in this library search the text buffer using a "sliding window" whose length equals the pattern length. First the left end of the window is aligned with the beginning of the text buffer, then the window is compared with the pattern. If a match is not found, the window is shifted to the right and the same procedure is repeated until the right end of the window moves past the end of the text buffer.</p> <p>This library supplies the following search functions:</p> <p>fastStrSearch() Optimally chooses the search algorithm based on the pattern size</p> <p>bmsStrSearch() Uses the efficient Boyer-Moore-Sunday search algorithm; may not be optimal for small patterns</p> <p>bfStrSearch() Uses the simple Brute Force search algorithm; best suited for small patterns</p> <p>To include this library, configure VxWorks with the <code>INCLUDE_STRING_SEARCH</code> component.</p>
INCLUDE FILE	strSearchLib.h

symLib

NAME	symLib – symbol table subroutine library
ROUTINES	symLibInit() – initialize the symbol table library symTblCreate() – create a symbol table symTblDelete() – delete a symbol table symAdd() – create and add a symbol to a symbol table, including a group number symRemove() – remove a symbol from a symbol table symFindByName() – look up a symbol by name symFindByNameAndType() – look up a symbol by name and type symByValueFind() – look up a symbol by value symByValueAndTypeFind() – look up a symbol by value and type symFindByValue() – look up a symbol by value symFindByValueAndType() – look up a symbol by value and type

symEach() – call a routine to examine each entry in a symbol table

DESCRIPTION

This library provides facilities for managing symbol tables. A symbol table associates a name and type with a value. A name is simply an arbitrary, null-terminated string. A symbol type is an unsigned char (typedef **SYM_TYPE**). A symbol value is a pointer. Though commonly used as the basis for object loaders, symbol tables may be used whenever efficient association of a value with a name is needed.

If you use the **symLib** subroutines to manage symbol tables local to your own applications, the values for **SYM_TYPE** objects are completely arbitrary; you can use whatever one-byte integers are appropriate for your application.

If you use the **symLib** subroutines to manipulate the VxWorks system symbol table (whose ID is recorded in the global variable **sysSymTbl**), the allowed values for **SYM_TYPE** are defined in **symbol.h**. They include macros for **SYM_UNDF**, **SYM_LOCAL**, **SYM_GLOBAL**, **SYM_ABS**, **SYM_TEXT**, **SYM_DATA**, **SYM_BSS**, and **SYM_COMM**. There are also macros defined there for determining whether a symbol is of a particular type. These include **SYM_IS_UNDF(symType)**, **SYM_IS_GLOBAL(symType)**, **SYM_IS_LOCAL(symType)**, **SYM_IS_TEXT(symType)**, etc. Using these macros helps to isolate the user of symbol tables from any changes that may be made to the way symbol types are handled internally within VxWorks.

USAGE

Tables are created with **symTblCreate()**, which returns a symbol table ID. This ID is used for all symbol table operations, including adding symbols, removing symbols, and searching for symbols. All operations on a symbol table are protected from re-entrancy problems by means of a mutual-exclusion semaphore in the symbol table structure. To ensure proper use of the symbol table semaphore, all symbol table accesses and operations should be performed using the API's provided by the **symLib** library. Symbol tables are deleted with **symTblDelete()**.

Symbols are added to a symbol table with **symAdd()**. Each symbol in the symbol table has a name, a value, a type and a reference. Symbols are removed from a symbol table with **symRemove()**.

Symbols can be accessed by either name or value. The routine **symFindByName()** searches the symbol table for a symbol with a specified name. The routine **symByValueFind()** finds a symbol with a specified value or, if there is no symbol with the same value, the symbol in the table with the largest value that is smaller than the specified value. Using this method, if an address is inside a function whose name is registered as a symbol, then the name of the function will be returned.

The routines **symFindByValue()** and **symFindByValueAndType()** are obsolete. They are replaced by the routines **symByValueFind()** and **symByValueAndTypeFind()** and will be removed in the next version of VxWorks.

Symbols in the symbol table are hashed by name into a hash table for fast look-up by name, e.g., by **symFindByName()**. The size of the hash table is specified during the creation of a

symbol table. Look-ups by value, e.g., **symByValueFind()**, must search the table linearly; these look-ups can therefore be much slower.

The routine **symEach()** allows every symbol in the symbol table to be examined by a user-specified function.

Name clashes occur when a symbol added to a table is identical in name and type to a previously added symbol. Whether or not symbol tables can accept name clashes is set by a parameter when the symbol table is created with **symTblCreate()**.

If name clashes are not allowed, **symAdd()** will return an error if there is an attempt to add a symbol with the same name and type as a symbol already in the symbol table.

If name clashes are allowed, adding multiple symbols with the same name and type will be permitted. In such cases, **symFindByName()** will return the value most recently added, although all versions of the symbol can be found using **symEach()**.

The system symbol table (**sysSymTbl**) allows name clashes.

See the VxWorks Programmer's Guide for more information about configuration, initialization, and use of the system symbol table.

INCLUDE FILES

symLib.h

ERRNOS

Routines from this library can return the following symbol-specific errors:

S_symLib_SYMBOL_NOT_FOUND

The requested symbol can not be found in the specified symbol table.

S_symLib_NAME_CLASH

A symbol of same name already exists in the specified symbol table (only when the name clash policy is selected at symbol table creation).

S_symLib_TABLE_NOT_EMPTY

The symbol table is not empty from its symbols, and then can not be deleted.

S_symLib_INVALID_SYMTAB_ID

The symbol table ID is invalid.

S_symLib_INVALID_SYM_ID_PTR.

The symbol table ID pointer is invalid.

S_symLib_INVALID_SYMBOL_NAME

The symbol name is invalid.

Note that other errors, not listed here, may come from libraries internally used by this library.

SEE ALSO

loadLib

symShow

NAME	symShow – symbol table show routines
ROUTINES	symShowInit() – initialize symbol table show routine symShow() – show the symbols of specified symbol table with matching substring
DESCRIPTION	This library provides a routine for showing symbol table information. The routine symShowInit() links the symbol table show facility into the VxWorks system. It is called automatically when this facility is configured into VxWorks by including the INCLUDE_SYM_TBL_SHOW component.
INCLUDE FILE	symLib.h
ERRNOS	Routines from this library can return the following symbol show specific errors: S_symLib_INVALID_SYMTAB_ID The specified symbol table ID is invalid.
SEE ALSO	symLib

sysLib

NAME	sysLib – system-dependent library
ROUTINES	sysClkConnect() – connect a routine to the system clock interrupt sysClkDisable() – turn off system clock interrupts sysClkEnable() – turn on system clock interrupts sysClkRateGet() – get the system clock rate sysClkRateSet() – set the system clock rate sysAuxClkConnect() – connect a routine to the auxiliary clock interrupt sysAuxClkDisable() – turn off auxiliary clock interrupts sysAuxClkEnable() – turn on auxiliary clock interrupts sysAuxClkRateGet() – get the auxiliary clock rate sysAuxClkRateSet() – set the auxiliary clock rate sysIntDisable() – disable a bus interrupt level sysIntEnable() – enable a bus interrupt level sysBusIntAck() – acknowledge a bus interrupt sysBusIntGen() – generate a bus interrupt sysMailboxConnect() – connect a routine to the mailbox interrupt sysMailboxEnable() – enable the mailbox interrupt

sysNvRamGet() – get the contents of non-volatile RAM
sysNvRamSet() – write to non-volatile RAM
sysModel() – return the model name of the CPU board
sysBspRev() – return the BSP version and revision number
sysHwInit() – initialize the system hardware
sysPhysMemTop() – get the address of the top of memory
sysMemTop() – get the address of the top of logical memory
sysToMonitor() – transfer control to the ROM monitor
sysProcNumGet() – get the processor number
sysProcNumSet() – set the processor number
sysBusTas() – test and set a location across the bus
sysScsiBusReset() – assert the RST line on the SCSI bus (Western Digital WD33C93 only)
sysScsiInit() – initialize an on-board SCSI port
sysScsiConfig() – system SCSI configuration
sysLocalToBusAdrs() – convert a local address to a bus address
sysBusToLocalAdrs() – convert a bus address to a local address
sysSerialHwInit() – initialize the BSP serial devices to a quiescent state
sysSerialHwInit2() – connect BSP serial device interrupts
sysSerialReset() – reset all SIO devices to a quiet state
sysSerialChanGet() – get the SIO_CHAN device associated with a serial channel
sysNanoDelay() – delay for specified number of nanoseconds

DESCRIPTION

This library provides board-specific routines.

NOTE: This is a generic reference entry for a BSP-specific library; this description contains general information only. For features and capabilities specific to the system library included in your BSP, see your BSP's reference entry for **sysLib**.

The file **sysLib.c** provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

Initialization functions

- initialize the hardware to a known state
- identify the system
- initialize drivers, such as SCSI or custom drivers

Memory/address space functions

- get the on-board memory size
- make on-board memory accessible to external bus
- map local and bus address spaces
- enable/disable cache memory
- set/get nonvolatile RAM (NVRAM)

- define board's memory map (optional)
- virtual-to-physical memory map declarations for processors with MMUs

Bus interrupt functions

- enable/disable bus interrupt levels
- generate bus interrupts

Clock/timer functions

- enable/disable timer interrupts
- set the periodic rate of the timer

Mailbox/location monitor functions

- enable mailbox/location monitor interrupts for VME-based boards

The **sysLib** library does not support every feature of every board; a particular board may have various extensions to the capabilities described here. Conversely, some boards do not support every function provided by this library. Some boards provide some of the functions of this library by means of hardware switches, jumpers, or PALs, instead of software-controllable registers.

Typically, most functions in this library are not called by the user application directly. The configuration modules **usrConfig.c** and **bootConfig.c** are responsible for invoking the routines at the appropriate time. Device drivers may use some of the memory mapping routines and bus functions.

INCLUDE FILES	sysLib.h
SEE ALSO	The VxWorks programmer guides, the BSP-specific reference entry for sysLib

syscallHookLib

NAME	syscallHookLib – SYSCALL Hook Support library
ROUTINES	syscallRegisterHookAdd() – add hook for system call group registration requests syscallRegisterHookDelete() – delete a previously added registration hook. syscallEntryHookAdd() – add a routine to be called on each system call entry syscallEntryHookDelete() – delete a previously added entry hook syscallExitHookAdd() – add a routine to be called on each system call exit syscallExitHookDelete() – delete a previously added exit hook
DESCRIPTION	This library provides routines for adding extensions to the VxWorks System Call library via hook routines. Hook routines can be added without modifying kernel code. The kernel

provides call-outs whenever system call groups are registered, and on entry and exit from system calls. Each hook type is represented as an array of function pointers. For each hook type, hook functions are called in the order they were added.

System call registration hooks are functions called when a call to **syscallGroupRegister()** is made. Registration hooks are passed the same parameters as those passed to **syscallGroupRegister()** itself. The hook functions are called in the order in which they were added, and must return either **OK** or **ERROR**. If the return value is anything other than **OK**, the group registration attempt is aborted, and **ERROR** is returned back to the user. This mechanism can be used to reject otherwise valid group registration requests.

System call entry hooks are functions called when a system call is made, and control is passed to the system call trap dispatcher in the kernel. The entry hook gets the same parameters as the system call dispatcher, and must return either **OK** or **ERROR**. If the return value is anything other than **OK**, the system call is aborted, and **ERROR** is returned back to the user. This mechanism can be used to reject otherwise valid system calls.

System call exit hooks are more informational in nature. Each exit hook is passed the return value from the system call. The hook function itself is not expected to return anything.

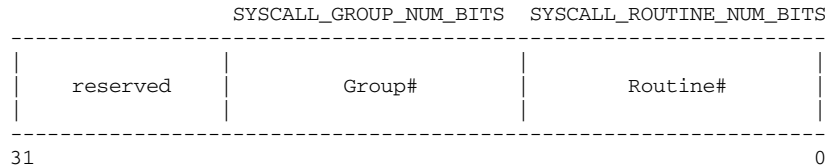
CONFIGURATION	To use the syscall hook support library, configure VxWorks with the INCLUDE_SYSCALL_HOOKS component and set parameter SYSCALL_HOOK_TBL_SIZE to a positive value (a value of at least 4 is recommended).
INCLUDE FILES	private/syscallLibP.h
SEE ALSO	syscallLib , the VxWorks programmer guides.

syscallLib

NAME	syscallLib – VxWorks System Call Infrastructure management library
ROUTINES	syscallGroupRegister() – register a system call group with the SCI syscallDispatch() – dispatch a system call request to its system call handler
DESCRIPTION	This is the System Call Infrastructure (SCI) library. The purpose of the SCI is to allow kernel routines to be invoked on behalf of applications. The kernel is organized in terms of various libraries that provide API's for users. Not all kernel API's are callable from user mode. A kernel library exporting its API to user-mode applications must first register with the SCI before any system calls are dispatched to it by the SCI.

SYSTEM CALL NUMBERS

A System Call Number (SCN) is a unique 32-bit number that identifies the kernel routine requested by the user. The SCI decodes this number to identify and call the kernel function to perform kernel work on behalf of the application. The SCN is organized as follows:



SYSTEM CALL GROUPS

A System Call Group is a logical grouping of related API's to be exposed to user-mode applications. VxWorks kernel code is organized into libraries that perform certain services and offer an API. This same library concept can be extended into the system call domain, for each kernel library wishing to export its API to user-space. Each such library can be treated as a System Call Group (SCG). Within the group, a library can have several functions that it wants to be callable via a system call.

Each System Call Group provides a table of routines that the SCI uses to call functions in it, in response to a system call. A System Call Group is represented in the system by a pointer to a table containing its routines, and a count of the number of routines it exports. A Group Table represents all SCG's in the system, and is an array of group structures.

It is not necessary to enforce a one-to-one mapping between a kernel library and a System Call Group number. More than one kernel library can share the same system call group without any impact on performance or complexity. However it should be kept in mind that it is convenient to group API's in logical order. Group numbering must also be unique across the operating system and also across future versions. Without this uniqueness rule, binary compatibility across OS versions will be broken.

The following is a visual representation for how **semLib** might be represented in the SCI:

semLib Group Tbl Entry	Routine Table for semLib		
pRoutineTbl ----+----->	pMethod	numArgs	methodName
numRoutines (4)	semTake	2	"semTake"
	semGive	1	"semGive"
	semFlush	1	"semFlush"
	semDelete	1	"semDelete"

SYSTEM CALL DISPATCHING

First, the group number is used to index into the Group Table. For example, if the group number for **semLib** is 5, the sixth entry of the Group Table (i.e., at offset 5) is accessed to get to the routine table for **semLib**. Then, the routine number of the SCN gives the index into the routine table, for the routine to be called. So **semFlush** (at offset 2 in the routine table) is the third routine exported by **semLib**. Joining the two together, the unique system call

number for semFlush is 0x0502. (This example is for expository purposes only. The **semLib** system calls actually belong to a system call group containing many other VxWorks-specific functions, not just semaphore operations.)

SYSTEM CALL ARGUMENT PASSING

System calls accept at most 8 arguments. Passing floating-point numbers or structures by value is not supported, and neither are variable argument list functions.

The total number and types of arguments must be limited to 8 native words for the architecture in question. A 64-bit argument of type long long is to be understood as 2 (possibly 3) 32-bit size arguments for a 32-bit architecture. Some architectures require 64-bit arguments to be in an even-odd numbered register pair, while some require them in an odd-even pair, while some others have no restriction at all. Alignment restrictions can mean having unused argument registers in the middle of an argument list. All such considerations taken together must to add up to 8 native-word-size arguments (including any padding registers).

SYSTEM CALL GROUP REGISTRATION

When the system is built, the Group Table is initialized with all System Call Groups known at compile time. As the other kernel libraries initialize themselves, they should call **syscallGroupRegister()** to register their API with the SCI. Registration involves populating the relevant entry in the Group Table - a pointer to the routine table, the number of routines exported, and a name string for show routines to display. Routine tables are allocated and populated by the respective kernel libraries.

Without registration, system calls made to a particular group will fail, returning **ERROR** and with errno set to **ENOSYS**. The dynamic nature of the registration process allows a kernel to be configured with just the needed functionality, without having to modify the system call infrastructure. The group table is statically allocated to hold space for all the defined groups.

CONFIGURATION The system call infrastructure management library is automatically included when the **INCLUDE_RTP** component is configured.

INCLUDE FILES **syscallLib.h** **private/syscallLibP.h**

syscallShow

NAME **syscallShow** – VxWorks System Call Infrastructure management library

ROUTINES **syscallShow()** – show registered System Call Groups, or a specific group
 syscallHookShow() – display all installed system call infrastructure hooks
 syscallMonitor() – monitor system call activity

DESCRIPTION	<p>This module implements a show facility for the System Call Infrastructure. The system call show routine can display either group-level information (level 0) or information about a specific group (level 1).</p> <p>It also provides a simple mechanism for monitoring system call activity along the lines of the UNIX truss utility.</p> <p>This library is initialized along with the RTP show library initialization. <code>INCLUDE_RTP_SHOW</code> and <code>INCLUDE_RTP</code> must both be defined to initialize this library.</p>
INCLUDE FILES	<code>syscallLib.h</code> <code>private/syscallLibP.h</code>

sysctl

NAME	<code>sysctl</code> – <code>sysctl</code> command
ROUTINES	<code>Sysctl()</code> – get or set values for kernel state variables from the C shell
DESCRIPTION	This module defines <code>Sysctl()</code> , a C shell utility that you can use to retrieve and configure run time parameters.
INCLUDE FILES	<code>sysctl.h</code>

tarLib

NAME	<code>tarLib</code> – UNIX tar compatible library
ROUTINES	<code>tarExtract()</code> – extract all files from a tar formatted tape <code>tarArchive()</code> – archive named file/dir onto tape in tar format <code>tarToc()</code> – display all contents of a tar formatted tape
DESCRIPTION	This library implements functions for archiving, extracting and listing of UNIX-compatible "tar" file archives. It can be used to archive and extract entire file hierarchies to/from archive files on local or remote disks, or directly to/from magnetic tapes.
CURRENT LIMITATIONS	This Tar utility does not handle MS-DOS file attributes, when used in conjunction with the MS-DOS file system. The maximum subdirectory depth supported by this library is 16, while the total maximum path name that can be handled by <code>tar</code> is limited at 100 characters.

INCLUDE FILES	none
SEE ALSO	dosFsLib

taskArchLib

NAME	taskArchLib – architecture-specific task management routines
ROUTINES	taskSRSet() – set the task status register (MC680x0, MIPS, x86) taskSRInit() – initialize the default task status register (MIPS)
DESCRIPTION	This library provides architecture-specific task management routines that set and examine architecture-dependent registers. For information about architecture-independent task management facilities, see the manual entry for taskLib .
NOTE	There are no application-level routines in taskArchLib for SimSolaris, SimNT or SH.
INCLUDE FILES	regs.h, taskArchLib.h
SEE ALSO	taskLib

taskHookLib

NAME	taskHookLib – task hook library
ROUTINES	taskCreateHookAdd() – add a routine to be called at every task create taskCreateHookDelete() – delete a previously added task create routine taskSwitchHookAdd() – add a routine to be called at every task switch taskSwitchHookDelete() – delete a previously added task switch routine taskDeleteHookAdd() – add a routine to be called at every task delete taskDeleteHookDelete() – delete a previously added task delete routine
DESCRIPTION	This library provides routines for adding extensions to the VxWorks tasking facility. To allow task-related facilities to be added to the system without modifying the kernel, the kernel provides call-outs every time a task is created, switched, or deleted. The call-outs allow additional routines, or "hooks," to be invoked whenever these events occur. The hook management routines below allow hooks to be dynamically added to and deleted from the current lists of create, switch, and delete hooks:

taskCreateHookAdd() and **taskCreateHookDelete()**

Add and delete routines to be called when a task is created.

taskSwitchHookAdd() and **taskSwitchHookDelete()**

Add and delete routines to be called when a task is switched.

taskDeleteHookAdd() and **taskDeleteHookDelete()**

Add and delete routines to be called when a task is deleted.

This facility is used by **dbgLib** to provide task-specific breakpoints and single-stepping. It is used by **taskVarLib** for the "task variable" mechanism. It is also used by **fppLib** for floating-point coprocessor support.

CONFIGURATION To use the task hook library, configure VxWorks with the **INCLUDE_TASK_HOOKS** component.

NOTE It is possible to have dependencies among task hook routines. For example, a delete hook may use facilities that are cleaned up and deleted by another delete hook. In such cases, the order in which the hooks run is important. VxWorks runs the create and switch hooks in the order in which they were added, and runs the delete hooks in reverse of the order in which they were added. Thus, if the hooks are added in "hierarchical" order, such that they rely only on facilities whose hook routines have already been added, then the required facilities will be initialized before any other facilities need them, and will be deleted after all facilities are finished with them.

VxWorks facilities guarantee this by having each facility's initialization routine first call any prerequisite facility's initialization routine before adding its own hooks. Thus, the hooks are always added in the correct order. Each initialization routine protects itself from multiple invocations, allowing only the first invocation to have any effect.

Task create hooks need to consider the ownership of any Wind objects, e.g. watchdog timers, semaphores, etc. created in the hook routine. Since create hook routines execute in the context of the creator task, new Wind objects will be owned by the creator task's RTP. It may be necessary to assign the ownership of these objects to the new task's RTP. This will prevent unexpected object reclamation from occurring if and when the RTP of the creator task terminates.

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

Switch hooks also need to be aware of the following restrictions:

- Do not assume any VM context is current other than the kernel context (as per ISRs).
- Do not rely on knowledge of the current task or invoke any function that relies on this information, e.g. **taskIdSelf()**.
- Do not rely on **taskIdVerify (pOldTcb)** to determine if a delete hook, if any, has already executed for the self-destructing task case. Instead, some other state

information needs to be changed, e.g. NULL'ing of a pointer, in the delete hook to be detected by the switch hook.

INCLUDE FILES **taskHookLib.h**

SEE ALSO **dbgLib, fppLib, taskLib, taskVarLib**, the VxWorks programmer guide.

taskHookShow

NAME **taskHookShow** – task hook show routines

ROUTINES **taskHookShowInit()** – initialize the task hook show facility
taskCreateHookShow() – show the list of task create routines
taskSwitchHookShow() – show the list of task switch routines
taskDeleteHookShow() – show the list of task delete routines

DESCRIPTION This library provides routines which summarize the installed kernel hook routines. There is one routine dedicated to the display of each type of kernel hook: task operation, task switch, and task deletion.

CONFIGURATION The routine **taskHookShowInit()** links the task hook show facility into the VxWorks system. It is called automatically when the task hook show facility is configured into VxWorks using the **INCLUDE_TASK_HOOK_SHOW** component.

INCLUDE FILES **taskHookLib.h**

SEE ALSO **taskHookLib**, *VxWorks Programmer's Guide: Basic OS*

taskInfo

NAME **taskInfo** – task information library

ROUTINES **taskOptionsSet()** – change task options
taskOptionsGet() – examine task options
taskRegsGet() – get a task's registers from the TCB
taskRegsSet() – set a task's registers
taskName() – get the name associated with a task ID
taskIdDefault() – set the default task ID
taskIsReady() – check if a task is ready to run

taskIsSuspended() – check if a task is suspended
taskIsStopped() – check if a task is stopped by the debugger
taskIsPended() – check if a task is pended
taskPriNormalGet() – get the normal priority of the task
taskIdListGet() – get a list of active task IDs
taskNameToId() – look up the task ID associated with a task name

DESCRIPTION This library provides a programmatic interface for obtaining task information.

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines **taskOptionsGet()**, **taskRegsGet()**, **taskName()**, **taskNameToId()**, **taskIsReady()**, **taskIsSuspended()**, **taskPriNormalGet()**, and **taskIdListGet()** are used to obtain task information. Three routines -- **taskOptionsSet()**, **taskRegsSet()**, and **taskIdDefault()** -- provide programmatic access to debugging features.

The chief drawback of using task information is that tasks may change their state between the time the information is gathered and the time it is utilized. Information provided by these routines should therefore be viewed as a snapshot of the system, and not relied upon unless the task is consigned to a known state, such as suspended.

Task management and control routines are provided by **taskLib**. Higher-level task information display routines are provided by **taskShow**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES taskLib.h

SEE ALSO taskLib, taskShow, taskHookLib, taskVarLib, semLib, kernelLib, the VxWorks programmer guides.

taskLib

NAME taskLib – task management library

ROUTINES taskInitExcStk() – initialize a task with stacks at specified addresses

taskActivate() – activate a task that has been initialized
taskSuspend() – suspend a task
taskResume() – resume a task
taskPrioritySet() – change the priority of a task
taskPriorityGet() – examine the priority of a task
taskLock() – disable task rescheduling
taskUnlock() – enable task rescheduling
taskSafe() – make the calling task safe from deletion
taskUnsafe() – make the calling task unsafe from deletion
taskDelay() – delay a task from executing
taskIdSelf() – get the task ID of a running task
taskIdVerify() – verify the existence of a task
taskTcb() – get the task control block for a task ID
taskStackAllot() – allot memory from a task's exception stack
taskCpuAffinitySet() – set the CPU affinity of a task
taskCpuAffinityGet() – get the CPU affinity of a task
taskCpuLock() – disable local CPU task rescheduling
taskCpuUnlock() – enable local CPU task rescheduling
taskSpawn() – spawn a task
taskCreate() – allocate and initialize a task without activation
taskInit() – initialize a task with a stack at a specified address
exit() – exit a task (ANSI)
taskExit() – exit a task
taskDelete() – delete a task
taskDeleteForce() – delete a task without restriction
taskRestart() – restart a task

DESCRIPTION This library provides the interface to the VxWorks task management facilities. Task control services are provided by the VxWorks kernel, which is comprised of **kernelLib**, **taskLib**, **semLib**, **tickLib**, **msgQLib**, and **wdLib**. Programmatic access to task information and debugging features is provided by **taskInfo**. Higher-level task information display routines are provided by **taskShow**.

TASK CREATION Tasks are created with the general-purpose routine **taskSpawn()**. Task creation consists of the following: allocation of memory for the stack and task control block (**WIND_TCB**), initialization of the **WIND_TCB**, and activation of the **WIND_TCB**. Special needs may require the use of the lower-level routines **taskInit()** and **taskActivate()**, which are the underlying primitives of **taskSpawn()**.

Tasks in VxWorks execute in the most privileged state of the underlying architecture. In a shared address space, processor privilege offers no protection advantages and actually hinders performance.

There is no limit to the number of tasks created in VxWorks, as long as sufficient memory is available to satisfy allocation requirements.

The routine **sp()** is provided in **usrLib** as a convenient abbreviation for spawning tasks. It calls **taskSpawn()** with default parameters.

TASK DELETION

If a task exits its "main" routine, specified during task creation, the kernel implicitly calls **exit()** to delete the task. Tasks can be explicitly deleted with the **taskDelete()** or **exit()** routine.

Task deletion must be handled with extreme care, due to the inherent difficulties of resource reclamation. Deleting a task that owns a critical resource can cripple the system, since the resource may no longer be available. Simply returning a resource to an available state is not a viable solution, since the system can make no assumption as to the state of a particular resource at the time a task is deleted.

The solution to the task deletion problem lies in deletion protection, rather than overly complex deletion facilities. Tasks may be protected from unexpected deletion using **taskSafe()** and **taskUnsafe()**. While a task is safe from deletion, deleters will block until it is safe to proceed. Also, a task can protect itself from deletion by taking a mutual-exclusion semaphore created with the **SEM_DELETE_SAFE** option, which enables an implicit **taskSafe()** with each **semTake()**, and a **taskUnsafe()** with each **semGive()** (see **semMLib** for more information). Many VxWorks system resources are protected in this manner, and application designers may wish to consider this facility where dynamic task deletion is a possibility.

The **sigLib** facility may also be used to allow a task to execute clean-up code before actually expiring.

TASK STACKS

In VxWorks every task has two stacks. The stack used for normal execution is simply called *stack*. The *exception stack* is the second stack. It is used during the processing of an exception, and in the case of tasks that run in real time processes, the exception stack is also used for the processing of a system call into the kernel. VxWorks manages the switching required between stacks without user intervention. There are a few APIs provided by this library that require users to be aware of this concept though.

TASK CONTROL

Tasks are manipulated by means of an ID that is returned when a task is created. VxWorks uses the convention that specifying a task ID of **NULL** in a task control function signifies the calling task.

The following routines control task state: **taskResume()**, **taskSuspend()**, **taskDelay()**, **taskRestart()**, **taskPrioritySet()**, and **taskRegsSet()**.

TASK SCHEDULING

VxWorks schedules tasks on the basis of priority. Tasks may have priorities ranging from 0, the highest priority, to 255, the lowest priority. The priority of a task in VxWorks is dynamic, and an existing task's priority can be changed using **taskPrioritySet()**.

TASK CPU AFFINITY

Task CPU affinity allows users to specify the CPU on which a task is to be run. While the APIs used to do this are available in both the uniprocessor version of VxWorks and VxWorks SMP, they really only have an effect in SMP systems. CPU affinity is meant to be a performance tuning tool that overrides the default VxWorks SMP scheduling behaviour, which is to dispatched tasks on any of the CPUs. CPU affinity can also be used to "stick" on a single CPU all tasks of an application that is not designed to be run in a concurrent environment. This effectively hides the SMP platform's concurrent execution characteristic from the application.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskInfo(), taskShow(), taskHookLib, taskVarLib, semLib, semMLib, kernelLib,**
VxWorks Programmer's Guide

taskOpen

NAME **taskOpen** – extended task management library

ROUTINES **taskOpenInit()** – initialize the task open facility
 taskOpen() – open a task
 taskClose() – close a task
 taskUnlink() – unlink a task

DESCRIPTION The extended task management library includes the APIs to open, close, and unlink tasks. Since these APIs did not exist in VxWorks 5.5, to prevent the functions from being included in the default image, they have been isolated from the general task management library.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU

(by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **taskLib.h**

SEE ALSO **msgQOpen**, **objOpen**, **semOpen**, **timerOpen**, the VxWorks, programmer guides.

taskRotate

NAME **taskRotate** – taskRotate functionality

ROUTINES **taskRotate()** – rotate ready queue for a given task priority

DESCRIPTION This library contains the implementation of the **taskRotate()** function. **taskRotate()** will rotate tasks in the VxWorks READY state for the specified priority.

The API **taskRotate()** will only be available when the **INCLUDE_TASK_ROTATE** component is included in the VxWorks configuration. This implementation also requires that VxWorks be built with the **INCLUDE_VX_TRADITIONAL_SCHEDULER** component, and that its **VX_TRAD_SCHED_CONSTANT_RDY_Q** parameter is set to **TRUE**. This is the normal (default) case for both command-line and project facility builds. See **usrKernel.c** for how this macro is used.

SMP CONSIDERATIONS

The **taskRotate()** API does not have defined behaviour in SMP, therefore any call to **taskRotate()** in a SMP system will return **ERROR**. In future releases, **taskRotate()** may be defined for a SMP system. Until then, the routine will return **ERROR**.

See the reference manual entry for **taskRotate()** for more information.

INCLUDE FILES **taskLib.h**

taskShow

NAME **taskShow** – task show routines

ROUTINES **taskShowInit()** – initialize the task show routine facility
 taskInfoGet() – get information about a task

taskShow() – display task information from TCBs
taskRegsShow() – display the contents of a task's registers
taskStatusString() – get a task's status as a string

DESCRIPTION	<p>This library provides routines to show task-related information, such as register values, task status etc.</p> <p>The taskShowInit() routine links the task show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:</p> <ul style="list-style-type: none">- If you use the configuration header files, define INCLUDE_SHOW_ROUTINES in config.h.- If you use the project facility, select INCLUDE_TASK_SHOW. <p>Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines taskInfoGet(), taskShow(), taskRegsShow() and taskStatusString() are used to display task information.</p> <p>The chief drawback of using task information is that tasks may change their state between the time the information is gathered and the time it is utilized. Information provided by these routines should therefore be viewed as a snapshot of the system, and not relied upon unless the task is consigned to a known state, such as suspended.</p> <p>Task management and control routines are provided by taskLib. Programmatic access to task information and debugging features is provided by taskInfo.</p>
INCLUDE FILES	taskLib.h
SEE ALSO	taskLib , taskInfo , taskHookLib , taskVarLib , semLib , kernelLib , the VxWorks programmer guides.

taskUtilLib

NAME	taskUtilLib – task utility library
ROUTINES	taskSpareNumAllot() – Allocate the first available spare field in the TCB taskSpareFieldSet() – set the spare field of a TCB taskSpareFieldGet() – get the spare field of a TCB
DESCRIPTION	This library provides a programmatic interface for obtaining and modifying task information.
INCLUDE FILES	taskLib.h

SEE ALSO **taskLib**

taskVarLib

NAME **taskVarLib** – task variables support library

ROUTINES **taskVarInit()** – initialize the task variables facility
taskVarAdd() – add a task variable to a task
taskVarDelete() – remove a task variable from a task
taskVarGet() – get the value of a task variable
taskVarSet() – set the value of a task variable
taskVarInfo() – get a list of task variables of a task

DESCRIPTION VxWorks provides a facility called "task variables," which allows 4-byte variables to be added to a task's context, and the variables' values to be switched each time a task switch occurs to or from the calling task. Typically, several tasks declare the same variable (4-byte memory location) as a task variable and treat that memory location as their own private variable. For example, this facility can be used when a routine must be spawned more than once as several simultaneous tasks.

The routines **taskVarAdd()** and **taskVarDelete()** are used to add or delete a task variable. The routines **taskVarGet()** and **taskVarSet()** are used to get or set the value of a task variable.

NOTE If you are using task variables in a task delete hook (see **taskHookLib**), refer to the reference entry for **taskVarInit()** for warnings on proper usage.

SMP CONSIDERATIONS This library is not available in VxWorks SMP. Use `__thread` variables instead.

INCLUDE FILES **taskVarLib.h**

SEE ALSO **taskHookLib**, the VxWorks programmer guides.

tc3c905VxbEnd

NAME **tc3c905VxbEnd** – 3Com 3c905/B/C VxBus END driver

ROUTINES **elPciRegister()** – register with the VxBus subsystem

DESCRIPTION

This module implements a driver for the 3Com Fast Etherlink XL PCI network interface cards. The Fast Etherlink XL family is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The original 3c905 adapter uses an external National Semiconductor DP83840A MII PHY, while the 3c905B and later adapters have an internal MII transceiver. Access to the PHYs on all boards is through a bit-bang MDIO interface.

The first generation of 3Com PCI ethernet cards were known as the Vortex family (3c590). These cards were based loosely on earlier 3Com ISA NIC designs. The Vortex supported bus master DMA, but did not use a descriptor ring-based API. It also included a PIO data transfer mechanism for compatibility with driver software for earlier devices.

The Vortex was followed by the Boomerang, aka the 3c905. The Boomerang introduced a descriptor-based DMA scheme which greatly reduced CPU overhead, however it also maintained compatibility with earlier Vortex devices.

The Boomerang was ultimately succeeded by the Cyclone, aka the 3c905B, which dropped the PIO compatibility interface entirely, added several I/O enhancements, included an integrated 10/100 PHY, and supported TCP/IP checksum offload.

The Cyclone was later followed by the Hurricane and Tornado chips, which maintained the existing Cyclone features and added remote management.

This driver supports all the cards in the Boomerang, Cyclone, Hurricane and Tornado families with 10/100 UTP and 10Mbps TPO interfaces. This includes the following:

*3c900-TPO 10Mbps 3c900B-TPO 10Mbps 3c905-TX 10/100Mbps 3c905B-TX 10/100Mbps
3c905C-TX 10/100Mbps 3c905CX-TXM 10/100Mbps 3c980-TX 10/100Mbps 3c980C-TX
10/100Mbps 3c918-TX 10/100Mbps 3c920-EMB 10/100Mbps 3c920-EMB-WNM 10/100Mbps
3cSOHO100-TX OfficeConnect 10/100Mbps 3c450-TX HomeConnect 10/100Mbps*

Cards with AUI/BNC ports, fiber optic interfaces, or 100baseT4 PHYs are not supported due to lack of available hardware for testing. (These cards are uncommon and are no longer being manufactured.)

The 3c90x uses a DMA descriptor scheme where each descriptor contains a 63-entry fragment list. Each fragment list entry is 8 bytes, which, along with 8 bytes for the status field and next pointer field, makes each descriptor 512 bytes in size. For RX handling, this driver only uses one fragment, so a special RX descriptor format is defined with just one fragment entry, for a total of 16 bytes.

RX DMA handling is reasonably straightforward: all descriptors are arranged into a circularly linked list, which the chip and host both traverse, using the **upload complete** bit in the status field to arbitrate access.

Transmit DMA handling is a little more complex: the driver must prepare a "download list" of descriptors containing outbound packets, and load the list address into the downlist pointer register each time it wants to initiate transmission. The last descriptor in the list must have a next pointer of 0, which will signal the chip to go idle once the last packet has been sent. It's possible to append additional descriptors to a transmission currently in progress, but the driver must pause the TX DMA engine first in order to avoid a race with the chip.

This driver uses the checksum offload feature available in all Cyclone, Hurricane and Tornado adapters for both RX and TX. The chip supports checksum offloading for IP, TCP and UDP, for IPv4 packets only.

BOARD LAYOUT The 3Com Fast Etherlink XL family of chips are available on standalone PCI cards as well as integrated onto various system boards. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **elPciRegister()** function, which registers the driver with VxBus.

SMP CONSIDERATIONS

The 3Com Fast Etherlink XL uses a register bank switching scheme to provide access to some of its registers. The total register space is 64 bytes in size for Boomerang adapters and 128 bytes for Cyclone, Hurricane and Tornado adapters. The first 16 bytes form a window through which 8 different register banks are visible, depending on the window selection bits of the command/status register. There are various registers of interest scattered throughout the different banks, including the RX filter control, MAC control and the MDIO access registers. Luckily, all of these special registers are accessed only in task context. Therefore, any code that switches register banks is guarded using the device semaphore.

The 3Com documentation states that some commands take more than one clock cycle to complete, requiring the driver to poll the command completion bit. This makes accesses to the command register non-atomic. Consequently, all accesses to the command/status register (offset 0xe), some of which are done in interrupt context, are guarded with a spinlock.

INCLUDE FILES none

SEE ALSO vxBus, miiBus, **ifLib**, "3com 3c90x and 3c90xB NICs Technical Reference, <http://www.freebsd.org/~wpaul/3Com/3c90xb.pdf>"

tffsDrv

NAME tffsDrv – TrueFFS interface for VxWorks

ROUTINES **tffsDrv()** – initialize the TrueFFS system
tffsDevCreate() – create a TrueFFS block device suitable for use with dosFs
tffsDevOptionsSet() – set TrueFFS volume options
tffsDrvOptionsSet() – set TrueFFS volume options
tffsDevFormat() – format a flash device for use with TrueFFS
tffsRawio() – low level I/O access to flash components

DESCRIPTION

This module defines the routines that VxWorks uses to create a TrueFFS block device. Using this block device, dosFs can access a board-resident flash memory array or a flash memory card (in the PCMCIA slot) just as if it was a standard disk drive. Also defined in this file are functions that you can use to format the flash medium, as well as functions that handle the low-level I/O to the device.

To include TrueFFS for Tornado in a VxWorks image, you must edit your BSP's **config.h** and define **INCLUDE_TFFS**, or, for some hardware, **INCLUDE_PCMCIA**. If you define **INCLUDE_TFFS**, this configures **usrRoot()** to call **tffsDrv()**. If you defined **INCLUDE_PCMCIA**, the call to **tffsDrv()** is made from **pccardTffsEnabler()**. The call to **tffsDrv()** sets up the structures, global variables, and mutual exclusion semaphore needed to manage TrueFFS. This call to **tffsDrv()** also registers socket component drivers for each flash device found attached to the target.

These socket component drivers are not quite block devices, but they are an essential layer within TrueFFS. Their function is to manage the hardware interface to the flash device, and they are intelligent enough to handle formatting and raw I/O requests to the flash device. The other two layers within TrueFFS are known as the translation layer and the MTD (the Memory Technology Driver). The translation layer of TrueFFS implements the error recover and wear-leveling features of TrueFFS. The MTD implements the low-level programming (map, read, write, and erase) of the flash medium.

To implement the socket layer, each BSP that supports TrueFFS includes a **sysTffs.c** file. This file contains the code that defines the socket component driver. This file also contains a set of defines that you can use to configure which translation layer modules and MTDs are included in TrueFFS. Which translation layer modules and MTDs you should include depends on which types of flash devices you need to support. Currently, there are three basic flash memory technologies, NAND-based, NOR-based, and SSFDC. Within **sysTffs.c**, define:

INCLUDE_TL_NFTL

To include the NAND-based translation layer module.

INCLUDE_TL_FTL

To include the NOR-based translation layer module.

INCLUDE_TL_SSFDC

To include the SSFDC-appropriate translation layer module.

To support these different technologies, TrueFFS ships with three different implementations of the translation layer. Optionally, TrueFFS can include all three modules. TrueFFS later binds the appropriate translation layer module to the flash device when it registers a socket component driver for the device.

Within these three basic flash device categories there are still other differences (largely manufacturer-specific). These differences have no impact on the translation layer. However, they do make a difference for the MTD. Thus, TrueFFS ships with eight different MTDs that can support a variety of flash devices from Intel, Sharp, Samsung, National, Toshiba, AMD, and Fujitsu. Within **sysTffs.c**, define:

INCLUDE_MTD_I28F016

For Intel 28f016 flash devices.

INCLUDE_MTD_I28F008

For Intel 28f008 flash devices.

INCLUDE_MTD_I28F008_BAJA

For Intel 28f008 flash devices on the Heurikon Baja 4000.

INCLUDE_MTD_AMD

For AMD, Fujitsu: 29F0{40,80,16} 8-bit flash devices.

INCLUDE_MTD_CDSN

For Toshiba, Samsung: NAND CDSN flash devices.

INCLUDE_MTD_DOC2

For Toshiba, Samsung: NAND DOC flash devices.

INCLUDE_MTD_CFISCS

For CFI/SCS flash devices.

INCLUDE_MTD_WAMD

For AMD, Fujitsu 29F0{40,80,16} 16-bit flash devices.

The socket component driver and the MTDs are provided in source form. If you need to write your own socket driver or MTD, use these working drivers as a model for your own.

EXTERNALLY CALLABLE ROUTINES

Most of the routines defined in this file are accessible through the I/O system only. However, four routines are callable externally. These are: **tfFsDrv()**, **tfFsDevCreate()**, **tfFsDevFormat()**, and **tfFsRawio()**.

The first routine called from this library must be **tfFsDrv()**. Call this routine exactly once. Normally, this is handled automatically for you from within **usrRoot()**, if **INCLUDE_TFFS** is defined, or from within **pccardTffsEnabler()**, if **INCLUDE_PCMCIA** is defined.

Internally, this call to **tfFsDrv()** registers socket component drivers for all the flash devices connected to your system. After registering a socket component driver for the device, TrueFFS can support calls to **tfFsDevFormat()** or **tfFsRawio()**. However, before you can mount dosFs on the flash device, you must call **tfFsDevCreate()**. This call creates a block device on top of the socket component driver, but does not mount dosFs on the device. Because mounting dosFs on the device is what you will want to do most of the time, the **sysTffs.c** file defines a helper function, **usrTffsConfig()**. Internally, this function calls **tfFsDevCreate()** and then does everything necessary (such as calling the **dosFsDevInit()** routine) to mount dosFs on the resulting block device.

LOW LEVEL I/O

Normally, you should handle your I/O to the flash device using dosFs. However, there are situations when that level of indirection is a problem. To handle such situations, this library defines **tfFsRawio()**. Using this function, you can bypass both dosFs and the TrueFFS translation services to program the flash medium directly.

However, you should not try to program the flash device directly unless you are intimately familiar with the physical limits of your flash device as well as with how TrueFFS formats the flash medium. Otherwise you risk not only corrupting the medium entirely but permanently damaging the flash device.

If all you need to do is write a boot image to the flash device, use the **tfFsBootImagePut()** utility instead of **tfFsRawio()**. This function provides safer access to the flash medium.

IOCTL	This driver responds to all ioctl codes by setting a global error flag. Do not attempt to format a flash drive using ioctl calls.
INCLUDE FILES	tfFsDrv.h, fatlite.h

tickLib

NAME	tickLib – clock tick support library
ROUTINES	tickAnnounce() – announce a clock tick to the kernel tickSet() – set the value of the kernel's tick counter tickGet() – get the value of the kernel's tick counter tick64Set() – set the value of the kernel's tick counter in 64 bits tick64Get() – get the value of the kernel's tick counter as a 64 bit value tickAnnounceHookAdd() – add a hook routine to be called at each tick interrupt
DESCRIPTION	<p>This library is the interface to the VxWorks kernel routines that announce a clock tick to the kernel, get the current time in ticks, and set the current time in ticks.</p> <p>Kernel facilities that rely on clock ticks include taskDelay(), wdStart(), kernelTimeSlice(), and semaphore timeouts. In each case, the specified timeout is relative to the current time, also referred to as "time to fire." Relative timeouts are not affected by calls to tickSet(), which only changes absolute time. The routines tickSet() and tickGet() keep track of absolute time in isolation from the rest of the kernel.</p> <p>Time-of-day clocks or other auxiliary time bases are preferable for lengthy timeouts of days or more. The accuracy of such time bases is greater, and some external time bases even calibrate themselves periodically.</p>
SMP CONSIDERATIONS	<p>All of the tickxxSet/tickxxGet APIs in this module are spinlock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs.</p>
INCLUDE FILES	tickLib.h

SEE ALSO **kernelLib**, **taskLib**, **semLib**, **wdLib**, The VxWorks Programmer's Guide

timerLib

NAME	timerLib – timer library (POSIX)
ROUTINES	timer_cancel() – cancel a timer timer_connect() – connect a user routine to the timer signal timer_create() – allocate a timer using the specified clock for a timing base (POSIX) timer_delete() – remove a previously created timer (POSIX) timer_gettime() – get the remaining time before expiration and the reload value (POSIX) timer_getoverrun() – return the timer expiration overrun (POSIX) timer_settime() – set the time until the next expiration and arm timer (POSIX) nanosleep() – suspend the current task until the time interval elapses (POSIX) sleep() – delay for a specified amount of time alarm() – set an alarm clock for delivery of a signal timer_modify() – modify a timer
DESCRIPTION	<p>This library provides a timer interface, as defined in the IEEE standard, POSIX 1003.1b.</p> <p>Timers are mechanisms by which tasks signal themselves after a designated interval. Timers are built on top of the clock and signal facilities. The clock facility provides an absolute time-base. Standard timer functions simply consist of creation, deletion and setting of a timer. When a timer expires, sigaction() (see sigLib) must be in place in order for the user to handle the event. The "high resolution sleep" facility, nanosleep(), allows sub-second sleeping to the resolution of the clock.</p> <p>The clockLib library should be installed and clock_settime() set before the use of any timer routines.</p>
CONFIGURATION	To use the POSIX timer library, configure VxWorks with the INCLUDE_POSIX_TIMERS component.
ADDITIONS	Two non-POSIX functions are provided for user convenience: timer_cancel() quickly disables a timer by calling timer_settime() . timer_connect() easily hooks up a user routine by calling sigaction() .
CLARIFICATIONS	<p>The task creating a timer with timer_create() will receive the signal no matter which task actually arms the timer.</p> <p>When a timer expires and the task has previously exited, logMsg() indicates the expected task is not present. Similarly, logMsg() indicates when a task arms a timer without</p>

installing a signal handler. Timers may be armed but not created or deleted at interrupt level.

As specified by the POSIX standard, the **sleep()** prototype is defined in **unistd.h**.

IMPLEMENTATION	The actual clock resolution is hardware-specific and in many cases is 1/60th of a second. This is less than <code>_POSIX_CLOCKRES_MIN</code> , which is defined as 20 milliseconds (1/50th of a second).
INCLUDE FILES	timers.h , unistd.h
SEE ALSO	clockLib , sigaction() , POSIX 1003.1b documentation

timerOpen

NAME	timerOpen – extended timer library
ROUTINES	timerOpenInit() – initialize the timer open facility timer_open() – open a timer timer_close() – close a named timer timer_unlink() – unlink a named timer
DESCRIPTION	The extended timer library includes the APIs to open, close, and unlink timers. Since these APIs did not exist in VxWorks 5.5, to prevent the functions from being included in the default image, they have been isolated from the general timer library.
SMP CONSIDERATIONS	Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling intCpuLock()) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.
INCLUDE FILES	timerLib.h
SEE ALSO	msgQOpen , objOpen , semOpen , taskOpen , the VxWorks, programmer guides.

timerShow

NAME	timerShow – POSIX timer show library
ROUTINES	timerShowInit() – initialize the timer show routine facility timer_show() – show information on a specified timer
DESCRIPTION	This Library contains the routine timer_show() , which displays a "snap shot" of a specified timer. The "snap shot" displays information about the timer at the time the function is called. The information displayed by timer_show() is intended for debugging purposes only.
CONFIGURATION	The routines in this library are included if the INCLUDE_POSIX_TIMER_SHOW component is configured into VxWorks. INCLUDE_POSIX_TIMER_SHOW requires the component INCLUDE_POSIX_TIMER .
INCLUDE FILES	time.h

timexLib

NAME	timexLib – execution timer facilities
ROUTINES	timexInit() – include the execution timer library timexClear() – clear the list of function calls to be timed timexFunc() – specify functions to be timed timexHelp() – display synopsis of execution timer facilities timex() – time a single execution of a function or functions timexN() – time repeated executions of a function or group of functions timexPost() – specify functions to be called after timing timexPre() – specify functions to be called prior to timing timexShow() – display the list of function calls to be timed
EXAMPLES	<p>The routine timex() can be used to obtain the execution time of a single routine:</p> <pre>-> timex myFunc, myArg1, myArg2, ...</pre> <p>The routine timexN() calls a function repeatedly until a 2% or better tolerance is obtained:</p> <pre>-> timexN myFunc, myArg1, myArg2, ...</pre> <p>The routines timexPre(), timexPost(), and timexFunc() are used to specify a list of functions to be executed as a group:</p> <pre>-> timexPre 0, myPreFunc1, preArg1, preArg2, ...</pre>

```

-> timexPre 1, myPreFunc2, preArg1, preArg2, ...

-> timexFunc 0, myFunc1, myArg1, myArg2, ...
-> timexFunc 1, myFunc2, myArg1, myArg2, ...
-> timexFunc 2, myFunc3, myArg1, myArg2, ...

-> timexPost 0, myPostFunc, postArg1, postArg2, ...

```

The list is executed by calling **timex()** or **timexN()** without arguments:

```

-> timex

or

-> timexN

```

In this example, *myPreFunc1* and *myPreFunc2* are called with their respective arguments. *myFunc1*, *myFunc2*, and *myFunc3* are then called in sequence and timed. If **timexN()** was used, the sequence is called repeatedly until a 2% or better error tolerance is achieved. Finally, *myPostFunc* is called with its arguments. The timing results are reported after all post-timing functions are called.

NOTES

The timings measure the execution time of the routine body, without the usual subroutine entry and exit code (usually LINK, UNLINK, and RTS instructions). Also, the time required to set up the arguments and call the routines is not included in the reported times. This is because these timing routines automatically calibrate themselves by timing the invocation of a null routine, and thereafter subtracting that constant overhead.

Also note that the measurement method is not immune to scheduling events. Should the timed tasks be stopped, pended or interrupted during measurement, the final data will take that into account. This is because the clock is still counting while the tasks are inactive.

SMP CONSIDERATIONS

On SMP systems, task migration also influence timings (see note about precision above). The VxWorks CPU affinity mechanism can be used to suppress that bias if deemed necessary (refer to the **taskLib** documentation for more information about CPU affinity).

INCLUDE FILES **timexLib.h**

SEE ALSO **spyLib**

tlsLib

NAME **tlsLib** – thread local storage support library

ROUTINES **tlsTaskInit()** – Thread Local Storage init routine

DESCRIPTION

The Thread Local Storage library provides VxWorks support for task specific variables. A task specific variable is declared using the `__thread` storage class keyword. For example, you can declare task specific variables like this:

```
__thread UINT32          myInteger;  
__thread char            myBuffer [10];
```

or add a reference to an external `__thread` variable like this:

```
extern __thread UINT32    myInteger;
```

When a VxWorks task is created, memory used to store `__thread` variables located in the VxWorks image (including user applications linked with the vxWorks image) for this specific task is allocated from the task exception stack. If your VxWorks image does contain a lot of `__thread` variables or if the size of `__thread` variables is big you may have to increase the size of the task exception stack in your VxWorks configuration.

The memory for `__thread` variables of a Downloadable Kernel Module (DKM) is allocated using the VxWorks memory manager; the memory required to store a DKM's `__thread` variables for a specific task is allocated only when this task makes access to one of the `__thread` variables of this modules. Because of this, accessing a `__thread` variable for the first time will not be deterministic. To avoid this, the entry point of the task can call **`tlsTaskInit()`** routine (this will allocate memory for all `__thread` variables of all DKMs) or simply access one of the DKM `__thread` variables (this will allocate memory only for the `__thread` variables of the DKM where is located the accessed `__thread` variable). The first solution will preserve determinism when accessing `__thread` variables for all DKMs in the system (that were loaded when **`tlsTaskInit()`** routine was called), the second solution will preserve determinism only when accessing a `__thread` variable of the same DKM (but will allocate memory only for `__thread` variables of this DKM).

When a task is deleted, all the memory it has used to manage and store `__thread` variables will be released.

When a DKM is unloaded, the memory used by a task to handle this module's `__thread` variables will be released only when the task will exit or when the task will try to access one of the DKM's `__thread` variables again. Note that when a task tries to access a `__thread` variable of a module that does not exist (because it has been unloaded for example), then a fatal ED&R event will be injected (if ED&R support is enabled) and the task will be stopped by the ED&R fatal policy handler.

`__thread` variables should not be accessed from an interrupt service routine. Accessing a `__thread` variable from an interrupt service routine may lead to unpredictable results.

INCLUDE FILES

`tlsLib.h`

SEE ALSO

trgLib

NAME	trgLib – trigger events control library
ROUTINES	trgLibInit() – initialize the triggering library trgWorkQReset() – Resets the trigger work queue task and queue trgAdd() – add a new trigger to the trigger list trgReset() – Reset a trigger in the trigger list trgDelete() – delete a trigger from the trigger list trgOn() – set triggering on trgOff() – set triggering off trgEnable() – enable a trigger trgDisable() – turn a trigger off trgChainSet() – chains two triggers trgEvent() – trigger a user-defined event
DESCRIPTION	<p>This library provides the interface for triggering events. The routines provide tools for creating, deleting, and controlling triggers. However, in most cases it is preferable to use the GUI to create and manage triggers, since all order and dependency factors are automatically accounted for there.</p> <p>The event types are defined as in the Wind River System Viewer. Triggering and the System Viewer share the same instrumentation points. Furthermore, one of the main uses of triggering is to start and stop System Viewer instrumentation. Triggering is started by the routine trgOn(), which sets the shared variable evtAction. Once the variable is set, when an instrumented point is hit, trgCheck() is called. The routine looks for triggers that apply to this event. The routine trgOff() stops triggering. The routine trgEnable() enables a specific trigger that was previously disabled with trgDisable(). (At creation time all triggers are enabled by default.) This routine also checks the number of triggers currently enabled, and when this is zero, it turns triggering off.</p>
NOTE	It is important to create a trigger before calling trgOn() . trgOn() checks the trigger list to see if there is at least one trigger there, and if not, it exits without setting evtAction .
INCLUDE FILES	trgLibP.h
SEE ALSO	<i>Wind River System Viewer User's Guide</i>

trgShow

NAME	trgShow – trigger show routine
------	---------------------------------------

ROUTINES	trgShowInit() – initialize the trigger show facility trgShow() – show trigger information
DESCRIPTION	<p>This library provides routines to show event triggering information, such as list of triggers, associated actions, trigger states, and so on.</p> <p>The routine trgShowInit() links the triggering show facility into the VxWorks system. It is called automatically when INCLUDE_TRIGGER_SHOW is defined.</p>
INCLUDE FILES	none
SEE ALSO	trgLib

tsecVxbEnd

NAME	tsecVxbEnd – Freescale TSEC VxBus END driver
ROUTINES	tsecRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver for the Motorola/Freescale Three Speed Ethernet Controller (TSEC) network interface. The TSEC supports 10, 100 and 1000Mbps operation over copper and fiber media.</p> <p>The TSEC is unusual in that it uses three different interrupt vectors: one for RX events, one for TX events and one for error events. The intention is to shave some cycles from the interrupt service path by jumping directly to the proper event handler routine instead of the driver having to determine the nature of pending events itself.</p> <p>Note that while this driver is VxBus-compliant, it does not use vxuDmaBufLib. The reason for this is that vxuDmaBufLib is technically only required for drivers for DMA-based devices that must be portable among multiple architectures (e.g. PCI or VMEbus adapters). The TSEC is not a standalone device: it only exists as an integrated component of certain MPC85xx and MPC83xx PowerPC CPUs. It is always big-endian, it is always cache-coherent (since we always enable the TSEC's snooping features), and it never needs bounce buffering or address translation. Given this, we may as well forgo the use of vxuDmaBufLib entirely, since using it will do nothing except add a bit of extra overhead to the packet processing paths.</p>
BOARD LAYOUT	The TSEC is directly integrated into the CPU. All configurations are jumperless.
EXTERNAL INTERFACE	<p>The driver provides a vxBus external interface. The only exported routine is the tsecRegister() function, which registers the driver with VxBus.</p>

The TSEC controller also supports jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of an buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface mottsec0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "mottsec", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

The TSEC controller also supports interrupt coalescing. This driver has coalescing support, which is disabled by default so that the **out of the box** configuration has the smallest interrupt latency. Coalescing can be enabled on a per-interface basis using parameter overides in the **hwconf.c** file, in the same way as jumbo frame support. In addition to turning the coalescing support on and off, the timeout and packet count values can be set:

```
{ "mottsec", 0, "coalesceEnable", VXB_PARAM_INT32, {(void *)1} }  
{ "mottsec", 0, "coalesceRxTicks", VXB_PARAM_INT32, {(void *)10} }  
{ "mottsec", 0, "coalesceRxPkts", VXB_PARAM_INT32, {(void *)8} }  
{ "mottsec", 0, "coalesceTxTicks", VXB_PARAM_INT32, {(void *)100} }  
{ "mottsec", 0, "coalesceTxPkts", VXB_PARAM_INT32, {(void *)16} }
```

If only the *coalesceEnable* property is set, the driver will use default timeout and packet count values as shown above. Specifying alternate values via the BSP will override the defaults.

INCLUDE FILES none

SEE ALSO vxBus, ifLib, miiBus, "Writing an Enhanced Network Driver", "MPC8560 PowerQUICC III Integrated Communications Processor Reference Manual",
http://www.freescale.com/files/32bit/doc/ref_manual/MPC8560RM.pdf

ttyDrv

NAME ttyDrv – provide terminal device access to serial channels

ROUTINES **ttyDrv()** – initialize the *tty* driver
 ttyDevCreate() – create a VxWorks device for a serial channel

DESCRIPTION This library provides the OS-dependent functionality of a serial device, including canonical processing and the interface to the VxWorks I/O system.

The BSP provides "raw" serial channels which are accessed via an **SIO_CHAN** data structure. These raw devices provide only low level access to the devices to send and receive characters. This library builds on that functionality by allowing the serial channels to be

accessed via the VxWorks I/O system using the standard read/write interface. It also provides the canonical processing support of **tyLib**.

CONFIGURATION	To use terminal device access to serial channels, configure VxWorks with the INCLUDE_TTY_DEV component. This library is initialized automatically when the INCLUDE_TTY_DEV component is configured in VxWorks.
INCLUDE FILES	ttyLib.h
SEE ALSO	tyLib , sioLib.h

tyLib

NAME	tyLib – <i>tty</i> driver support library
ROUTINES	tyLibInit() – initialize the <i>tty</i> library tyDevInit() – initialize the <i>tty</i> device descriptor tyDevRemove() – remove the <i>tty</i> device descriptor tyDevTerminate() – terminate the <i>tty</i> device descriptor tyAbortFuncSet() – set the abort function tyAbortSet() – change the abort character tyAbortGet() – get the abort character tyBackspaceSet() – change the backspace character tyDeleteLineSet() – change the line-delete character tyEOFSet() – change the end-of-file character tyEOFGet() – get the current end-of-file character tyMonitorTrapSet() – change the trap-to-monitor character tyIoctl() – handle device control requests tyWrite() – do a task-level write for a <i>tty</i> device tyRead() – do a task-level read for a <i>tty</i> device tyITx() – interrupt-level output tyIRd() – interrupt-level input tyXoffHookSet() – install a hardware flow control function
DESCRIPTION	This library provides routines used to implement drivers for serial devices. It provides all the necessary device-independent functions of a normal serial channel, including: <ul style="list-style-type: none">- ring buffering of input and output- raw mode- optional line mode with backspace and line-delete functions

- optional processing of X-on/X-off
- optional RETURN/LINEFEED conversion
- optional echoing of input characters
- optional stripping of the parity bit from 8-bit input
- optional special characters for shell abort and system restart

Most of the routines in this library are called only by device drivers. Functions that normally might be called by an application or interactive user are the routines to set special characters, **ty...Set()**.

USE IN SERIAL DEVICE DRIVERS

Each device that uses **tyLib** is described by a data structure of type **TY_DEV**. This structure begins with an I/O system device header so that it can be added directly to the I/O system's device list. A driver calls **tyDevInit()** to initialize a **TY_DEV** structure for a specific device and then calls **iosDevAdd()** to add the device to the I/O system. Prior to driver termination, if ever, the driver calls **tyDevTerminate()** to terminate a **TY_DEV** structure.

The call to **tyDevInit()** takes three parameters: the pointer to the **TY_DEV** structure to initialize, the desired size of the read and write ring buffers, and the address of a transmitter start-up routine. This routine will be called when characters are added for output and the transmitter is idle. Thereafter, the driver can call the following routines to perform the usual device functions:

- tyRead()**
user read request to get characters that have been input
- tyWrite()**
user write request to put characters to be output
- tyIoctl()**
user I/O control request
- tyIRd()**
interrupt-level routine to get an input character
- tyITx()**
interrupt-level routine to deliver the next output character

Thus, **tyRead()**, **tyWrite()**, and **tyIoctl()** are called from the driver's read, write, and I/O control functions. The routines **tyIRd()** and **tyITx()** are called from the driver's interrupt handler in response to receive and transmit interrupts, respectively.

Examples of using **tyLib** in a driver can be found in the source file(s) included by **tyCoDrv**. Source files are located in `src/drv/serial`.

TTY OPTIONS

A full range of options affects the behavior of *tty* devices. These options are selected by setting bits in the device option word using the **FIOSETOPTIONS** function in the **ioctl()**

routine (see "I/O Control Functions" below for more information). The following is a list of available options. The options are defined in the header file **ioLib.h**.

OPT_LINE

Selects line mode. A *tty* device operates in one of two modes: raw mode (unbuffered) or line mode. Raw mode is the default. In raw mode, each byte of input from the device is immediately available to readers, and the input is not modified except as directed by other options below. In line mode, input from the device is not available to readers until a NEWLINE character is received, and the input may be modified by backspace, line-delete, and end-of-file special characters. The max line length in line mode is set to **MAX_CANON**, defined in **limits.h**. This limit includes the end of line character.

OPT_ECHO

Causes all input characters to be echoed to the output of the same channel. This is done simply by putting incoming characters in the output ring as well as the input ring. If the output ring is full, the echoing is lost without affecting the input.

OPT_CRMOD

C language conventions use the NEWLINE character as the line terminator on both input and output. Most terminals, however, supply a RETURN character when the return key is hit, and require both a RETURN and a LINEFEED character to advance the output line. This option enables the appropriate translation: NEWLINES are substituted for input RETURN characters, and NEWLINES in the output file are automatically turned into a RETURN-LINEFEED sequence.

OPT_TANDEM

Causes the driver to generate and respond to the special flow control characters CTRL-Q and CTRL-S in what is commonly known as X-on/X-off protocol. Receipt of a CTRL-S input character will suspend output to that channel. Subsequent receipt of a CTRL-Q will resume the output. Also, when the VxWorks input buffer is almost full, a CTRL-S will be output to signal the other side to suspend transmission. When the input buffer is almost empty, a CTRL-Q will be output to signal the other side to resume transmission.

OPT_7_BIT

Strips the most significant bit from all bytes input from the device.

OPT_MON_TRAP

Enables the special monitor trap character, by default CTRL-X. When this character is received and this option is enabled, VxWorks will trap to the ROM resident monitor program. Note that this is quite drastic. All normal VxWorks functioning is suspended, and the computer system is entirely controlled by the monitor. Depending on the particular monitor, it may or may not be possible to restart VxWorks from the point of interruption. The default monitor trap character can be changed by calling **tyMonitorTrapSet()**.

OPT_ABORT

Enables the special shell abort character, by default CTRL-C. When this character is received and this option is enabled, the VxWorks shell is restarted. This is useful for

freeing a shell stuck in an unfriendly routine, such as one caught in an infinite loop or one that has taken an unavailable semaphore. For more information, see the VxWorks programmer guides.

OPT_TERMINAL

This is not a separate option bit. It is the value of the option word with all the above bits set.

OPT_RAW

This is not a separate option bit. It is the value of the option word with none of the above bits set.

I/O CONTROL FUNCTIONS

The *tty* devices respond to the following **ioctl()** functions. The functions are defined in the header **ioLib.h**.

FIOGETNAME

Gets the file name of the file descriptor and copies it to the buffer referenced to by *nameBuf*:

```
status = ioctl (fd, FIOGETNAME, &nameBuf);
```

This function is common to all file descriptors for all devices.

FIOSETOPTIONS, FIOOPTIONS

Sets the device option word to the specified argument. For example, the call:

```
status = ioctl (fd, FIOOPTIONS, OPT_TERMINAL);  
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL);
```

enables all the *tty* options described above, putting the device in a "normal" terminal mode. If the line protocol (**OPT_LINE**) is changed, the input buffer is flushed. The various options are described in **ioLib.h**.

FIOGETOPTIONS

Returns the current device option word:

```
options = ioctl (fd, FIOGETOPTIONS, 0);
```

FIONREAD

Copies to *nBytesUnread* the number of bytes available to be read in the device's input buffer:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

In line mode (**OPT_LINE** set), the FIONREAD function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines of just NEWLINES were in the input buffer, it would return the value 10 (5 characters + 5 lines).

FIONWRITE

Copies to *nBytes* the number of bytes queued to be output in the device's output buffer:

```
status = ioctl (fd, FIONWRITE, &nBytes);
```

FIOFLUSH

Discards all the bytes currently in both the input and the output buffers:

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOWFLUSH

Discards all the bytes currently in the output buffer:

```
status = ioctl (fd, FIOWFLUSH, 0);
```

FIORFLUSH

Discards all the bytes currently in the input buffers:

```
status = ioctl (fd, FIORFLUSH, 0);
```

FIOCANCEL

Cancels a read or write. A task blocked on a read or write may be released by a second task using this `ioctl()` call. For example, a task doing a read can set a watchdog timer before attempting the read; the auxiliary task would wait on a semaphore. The watchdog routine can give the semaphore to the auxiliary task, which would then use the following call on the appropriate file descriptor:

```
status = ioctl (fd, FIOCANCEL, 0);
```

FIOBAUDRATE

Sets the baud rate of the device to the specified argument. For example, the call:

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

Sets the device to operate at 9600 baud. This request has no meaning on a pseudo terminal.

FIOISATTY

Returns **TRUE** for a *tty* device:

```
status = ioctl (fd, FIOISATTY, 0);
```

FIOPROTOHOOK

Adds a protocol hook function to be called for each input character. *pfunction* is a pointer to the protocol hook routine which takes two arguments of type *int* and returns values of type **STATUS** (**TRUE** or **FALSE**). The first argument passed is set by the user via the `FIOPROTOARG` function. The second argument is the input character. If no further processing of the character is required by the calling routine (the input routine of the driver), the protocol hook routine *pFunction* should return **TRUE**. Otherwise, it should return **FALSE**:

```
status = ioctl (fd, FIOPROTOHOOK, pFunction);
```

FIOPROTOARG

Sets the first argument to be passed to the protocol hook routine set by `FIOPROTOHOOK` function:

```
status = ioctl (fd, FIOPROTOARG, arg);
```

FIORBUFSET

Changes the size of the receive-side buffer to *size*:

```
status = ioctl (fd, FIORBUFSET, size);
```

FIOWBUFSET

Changes the size of the send-side buffer to *size*:

```
status = ioctl (fd, FIOWBUFSET, size);
```

Any other **ioctl()** request will return an error and set the status to **S_ioLib_UNKNOWN_REQUEST**.

CONFIGURATION	To use the <i>tty</i> driver support library, configure VxWorks with the INCLUDE_TYLIB component.
INCLUDE FILES	tyLib.h , ioLib.h
SEE ALSO	ioLib , iosLib , tyCoDrv , the VxWorks programmer guides.

unShow

NAME	unShow – information display routines for AF_LOCAL
ROUTINES	unstatShow() – display all AF_LOCAL sockets
DESCRIPTION	<p>This library provides routines to display information for all sockets and statistics for the AF_LOCAL address family.</p> <p>The unShowInit() routine links the AF_LOCAL show facility into the VxWorks system. This is performed automatically if INCLUDE_UN_SHOW is defined. Components need to be added per-protocol as well, eg. INCLUDE_UN_COMP_SHOW for the COMP protocol unstatShow() support.</p>
INCLUDE FILES	none

unixDrv

NAME	unixDrv – UNIX-file disk driver (VxSim for Solaris)
ROUTINES	unixDrv() – install UNIX disk driver

unixDiskDevCreate() – create a UNIX disk device
unixDiskInit() – initialize a dosFs disk on top of UNIX

DESCRIPTION This driver emulates a VxWorks disk driver by using a virtual disk. The VxWorks disk appears under UNIX as a single file. The UNIX file name, and the size of the disk, may be specified during the **unixDiskDevCreate()** call.

USER-CALLABLE ROUTINES

The routine **unixDrv()** must be called to initialize the driver and the **unixDiskDevCreate()** routine is used to create devices.

CREATING UNIX DISKS

Before a UNIX disk can be used, it must be created. This is done with the **unixDiskDevCreate()** call. The format of this call is:

```
BLK_DEV *unixDiskDevCreate
(
    char      *unixFile,      /* name of the UNIX file to use      */
    int       bytesPerBlk,    /* number of bytes per block         */
    int       blksPerTrack,   /* number of blocks per track        */
    int       nBlocks        /* number of blocks on this device    */
)
```

The UNIX file must be pre-allocated separately. This can be done using the UNIX `mkfile(8)` command. Note that you have to create an appropriately sized file. For example, to create a UNIX file system that is used as a common floppy dosFs file system, you would issue the command:

```
mkfile 1440k /tmp/floppy.dos
```

This will create space for a 1.44 Meg DOS floppy (1474560 bytes, or 2880 512-byte blocks).

The *bytesPerBlk* parameter specifies the size of each logical block on the disk. If *bytesPerBlk* is zero, 512 is the default.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the UNIX disk. If *blksPerTrack* is zero, the count of blocks per track will be set to *nBlocks* (i.e., the disk will be defined as having only one track). UNIX disk devices typically are specified with only one track.

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero the size of the UNIX file specified, divided by the number of bytes per block, is used.

The formatting parameters (*bytesPerBlk*, *blksPerTrack*, and *nBlocks*) are critical only if the UNIX disk already contains the contents of a disk created elsewhere. In that case, the formatting parameters must be identical to those used when the image was created. Otherwise, they may be any convenient number.

Once the device has been created it still does not have a name or file system associated with it. This must be done by using the file system's device initialization routine (e.g., **dosFsDevInit()**). The dosFs file systems also provide make-file-system routines

(**dosFsMkfs()**), which may be used to associate a name and file system with the block device and initialize that file system on the device using default configuration parameters.

The **unixDiskDevCreate()** call returns a pointer to a block device structure (**BLK_DEV**). This structure contains fields that describe the physical properties of a disk device and specify the addresses of routines within the UNIX disk driver. The **BLK_DEV** structure address must be passed to the desired file system (**dosFs**, or **rawFs**) during the file system's device initialization or make-file-system routine. Only then is a name and file system associated with the device, making it available for use.

As an example, to create a 200KB disk, 512-byte blocks, and only one track, the proper call would be:

```
BLK_DEV *pBlkDev;

pBlkDev = unixDiskDevCreate ("/tmp/filesys1", 512, 400, 400, 0);
```

This will attach the UNIX file `/tmp/filesys1` as a block device.

A convenience routine, **unixDiskInit()**, is provided to do the **unixDiskDevCreate()** followed by either a **dosFsMkFs()** or **dosFsDevInit()**, whichever is appropriate.

The format of this call is:

```
BLK_DEV *unixDiskInit
(
    char * unixFile, /* name of the UNIX file to use */
    char * volName,  /* name of the dosFs volume to use */
    int   nBytes     /* number of bytes in dosFs volume */
)
```

This call will create the UNIX disk if required.

IMPORTANT NOTE This library is obsolete, but is kept for backward compatibility and is now replaced by virtual disk library. But note that the virtual disk library is not able to use unix disk created with **unixDiskDevCreate()** API.

INCLUDE FILES none

SEE ALSO **dosFsDevInit()**, **dosFsMkfs()**, **rawFsDevInit()**

unldLib

NAME **unldLib** – object module unloading library

ROUTINES **unldByModuleId()** – unload an object module by specifying a module ID
unldByNameAndPath() – unload an object module by specifying a name and path
unldByGroup() – unload an object module by specifying a group number

DESCRIPTION This library provides a facility for unloading *code modules* (see **loadLib** for a definition of *code modules*). Once a code module has been installed in the system using **loadLib**, it can be removed from the system by calling one of the **unldXxx()** routines in this library.

Unloading a code module involves performing the following actions:

- (1) Free the space allocated for the code module segments (text, data, and BSS), unless **loadModule()** was given load directives with specific segment addresses, in which case the user is responsible for freeing the space.
- (2) Remove all symbols associated with the object module from the symbol table.
- (3) Remove the code module descriptor, and its segment descriptors, from the code module list.

Once the code module is unloaded, any calls to routines in that module from other modules will fail unpredictably. In this case, the user is responsible for ensuring that no modules are unloaded that are used by other modules. **unldByModuleId()** checks the hooks created by the following routines to ensure none of the unloaded code is in use by a hook:

```
taskCreateHookAdd()  
taskDeleteHookAdd()  
taskSwapHookAdd()  
taskSwitchHookAdd()
```

However, **unldByModuleId()** *does not* check any hook added by a network component, by ED&R or the hooks created by the following routines:

```
excHookAdd()  
rebootHookAdd()  
moduleCreateHookAdd()
```

INCLUDE FILES unldLib.h, moduleLib.h

ERRNOS Routines from this library can return the following unloader-specific errors:

S_unldLib_TEXT_IN_USE

The text segment of the module which is being unloaded is still in use.

Note that other errors, not listed here, may come from libraries internally used by the unloader.

SEE ALSO loadLib, moduleLib

usrConfig

NAME	usrConfig – user-defined system configuration library
ROUTINES	usrInit() – user-defined system initialization routine usrRoot() – the root task usrClock() – user-defined system clock interrupt routine
DESCRIPTION	<p>This library is the WRS-supplied configuration module for VxWorks. It contains the root task, the primary system initialization routine, the network initialization routine, and the clock interrupt routine.</p> <p>The include file config.h includes a number of system-dependent parameters used in this file.</p> <p>In an effort to simplify the presentation of the configuration of vxWorks, this file has been split into smaller files. These additional configuration source files are located in <code>../src/config/usr[xxx].c</code> and are <code>#included</code> into this file below. This file contains the bulk of the code a customer is likely to customize.</p> <p>The module usrDepend.c contains checks that guard against unsupported configurations such as <code>INCLUDE_NFS</code> without <code>INCLUDE_RPC</code>. The module usrKernel.c contains the core initialization of the kernel which is rarely customized, but provided for information. The module usrNetwork.c now contains all network initialization code. Finally, the module usrExtra.c contains the conditional inclusion of the optional packages selected in configAll.h.</p> <p>The source code necessary for the configuration selected is entirely included in this file during compilation as part of a standard build in the board support package. No other make is necessary.</p>
INCLUDE FILES	config.h
SEE ALSO	The VxWorks programmer guides.

usrFdiskPartLib

NAME	usrFdiskPartLib – FDISK-style partition handler
ROUTINES	usrFdiskPartRead() – read an FDISK-style partition table usrFdiskPartCreate() – create an FDISK-like partition table on a disk usrFdiskPartShow() – parse and display partition data

DESCRIPTION

This module is provided as source code to accommodate various customizations of partition table handling, resulting from variations in the partition table format in a particular configuration. It is intended for use with dpartCbio partition manager.

This code supports both mounting MSDOS file systems and displaying partition tables written by MSDOS FDISK.exe or by any other MSDOS FDISK.exe compatible partitioning software.

The first partition table is contained within a hard drive's Master Boot Record (MBR) sector, which is defined as sector one, cylinder zero, head zero or logical block address zero.

The mounting and displaying routines within this code will first parse the MBR partition tables entries (defined below) and also recursively parse any "extended" partition tables, which may reside within another sector further into the hard disk. MSDOS file systems within extended partitions are known to those familiar with the MSDOS FDISK.exe utility as "Logical drives within the extended partition".

Here is a picture showing the layout of a single disk containing multiple MSDOS file systems:

```

+-----+
|<-----The entire disk----->|
|M
|B<---C:--->
|R      /---- First extended partition-----\
|      E<---D:---><--Rest of the ext part----->|
|P      x
|A      t      E<---E:--->E<Rest of the ext part->|
|R      x      x
|T      t      t<-----F:----->|
+-----+

```

(Ext == extended partition sector)

C: is a primary partiion

D:, E:, and F: are logical drives within the extended partition.

A MS-DOS partition table resides within one sector on a hard disk. There is always one in the first sector of a hard disk partitioned with FDISK.exe. There first partition table may contain references to "extended" partition tables residing on other sectors if there are multiple partitions. The first sector of the disk is the starting point. Partition tables are of the format:

Offset from
the beginning
of the sector

Description

0x1be	Partition 1 table entry (16 bytes)
0x1ce	Partition 2 table entry (16 bytes)
0x1de	Partition 3 table entry (16 bytes)
0x1ee	Partition 4 table entry (16 bytes)
0x1fe	Signature (0x55aa, 2 bytes)

Individual MSDOS partition table entries are of the format:

Offset	Size	Description
-----	----	-----

0x0	8 bits	boot type
0x1	8 bits	beginning sector head value
0x2	8 bits	beginning sector (2 high bits of cylinder#)
0x3	8 bits	beginning cylinder# (low order bits of cylinder#)
0x4	8 bits	system indicator
0x5	8 bits	ending sector head value
0x6	8 bits	ending sector (2 high bits of cylinder#)
0x7	8 bits	ending cylinder# (low order bits of cylinder#)
0x8	32 bits	number of sectors preceding the partition
0xc	32 bits	number of sectors in the partition

The Cylinder, Head and Sector values herein are not used, instead the 32-bit partition offset and size (also known as LBA addresses) are used exclusively to determine partition geometry.

If a non-partitioned disk is detected, in which case the 0'th block is a DosFs boot block rather than an MBR, the entire disk will be configured as partition 0, so that disks formatted with VxWorks and disks formatted on MS-DOS or Windows can be accepted interchangeably.

The `usrFdiskPartCreate()` will create a partition table with up to four partitions, which can be later used with `usrFdiskPartRead()` and `dpartCbio` to manage a partitioned disk on VxWorks.

However, it can not be guaranteed that this partition table can be used on another system due to several BIOS specific paramaters in the boot area. If interchangeability via removable disks is a requirement, partition tables should be created and volumes should be formatted on the other system with which the data is to be interchanged.

CAUTION	<p>The partition decode function is recursive, up to the maximum number of partitions expected, which is no more then 24.</p> <p>Sufficient stack space needs to be provided via <code>taskSpawn()</code> to accommodate the recursion level.</p>
INCLUDE FILES	none
SEE ALSO	<code>dpartCbio</code>

usrFsLib

NAME	<code>usrFsLib</code> – file system user interface subroutine library
ROUTINES	<code>cd()</code> – change the default directory <code>pwd()</code> – print the current default directory <code>mkdir()</code> – make a directory <code>rmdir()</code> – remove a directory

rm() – remove a file
copyStreams() – copy from/to specified streams
copy() – copy *in* (or *stdin*) to *out* (or *stdout*)
chkdsk() – perform consistency checking on a MS-DOS file system
dirList() – list contents of a directory (multi-purpose)
ls() – generate a brief listing of a directory
ll() – generate a long listing of directory contents
lsr() – list the contents of a directory and any of its subdirectories
llr() – do a long listing of directory and all its subdirectories contents
cp() – copy file into other file/directory.
mv() – mv file into other directory.
xcopy() – copy a hierarchy of files with wildcards
xdelete() – delete a hierarchy of files with wildcards
attrib() – modify MS-DOS file attributes on a file or directory
xattrib() – modify MS-DOS file attributes of many files
dosfsDiskFormat() – format a disk with dosFs
diskFormat() – format a disk with dosFs
hrfsDiskFormat() – format a disk with HRFS
diskInit() – initialize a file system on a block device
commit() – commit current transaction to disk.
ioHelp() – print a synopsis of I/O utility functions

DESCRIPTION This library provides user-level utilities for managing file systems. These utilities may be used from Host Shell, the Kernel Shell or from an application.

USAGE FROM HOST SHELL

Some of the functions in this library have counterparts of the same names built into the Host Shell (aka Windsh). The built-in functions perform similar functions on the Tornado host computer's I/O systems. Hence if one of such functions needs to be executed in order to perform any operation on the Target's I/O system, it must be preceded with an @ sign, e.g.:

```
-> @ls "/sd0"
```

will list the directory of a disk named "/sd0" on the target, while

```
-> ls "/tmp"
```

will list the contents of the "/tmp" directory on the host.

The target I/O system and the Host Shell running on the host, each have their own notion of current directory, which are not related, hence

```
-> pwd
```

will display the Host Shell current directory on the host file system, while

```
-> @pwd
```

will display the target's current directory on the target's console.

WILDCARDS Some of the functions herein support wildcard characters in argument strings where file or directory names are expected. The wildcards are limited to "*" which matches zero or more characters and "?" which matches any single characters. Files or directories with names beginning with a "." are not normally matched with the "*" wildcard.

DIRECTORY LISTING

Directory listing is implemented in one function **dirList()**, which can be accessed using one of these four front-end functions:

ls()

produces a short list of files

lsr()

is like **ls()** but ascends into subdirectories

ll()

produces a detailed list of files, with file size, modification date attributes etc.

llr()

is like **ll()** but also ascends into subdirectories

All of the directory listing functions accept a name of a directory or a single file to list, or a name which contain wildcards, which will result in listing of all objects that match the wildcard string provided.

INCLUDE FILES **usrLib.h**

SEE ALSO **ioLib**, **dosFsLib**, **netDrv**, **nfsLib**, **hrFsLib**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

usrLib

NAME **usrLib** – user interface subroutine library

ROUTINES

- help()** – print a synopsis of selected routines
- netHelp()** – print a synopsis of network routines
- w()** – print a summary of each task's pending information, task by task
- tw()** – print info about the object the given task is pending on
- shConfig()** – display or set the shell configuration
- strFree()** – free shell strings
- bootChange()** – change the boot line
- periodRun()** – call a function periodically
- period()** – spawn a task to call a function periodically
- repeatRun()** – call a function repeatedly
- repeat()** – spawn a task to call a function repeatedly

sp() – spawn a task with default parameters
checkStack() – print a summary of each task's stack usage
i() – print a summary of each task's TCB
ti() – print complete information from a task's TCB
show() – print information on a specified object
ts() – suspend a task
tr() – resume a task
td() – delete a task
version() – print VxWorks version information
m() – modify memory
d() – display memory
ld() – load an object module into memory
devs() – list all system-known devices
lkup() – list symbols
lkAddr() – list symbols whose values are near a specified value
mRegs() – modify registers
pc() – return the contents of the program counter
printErrno() – print the definition of a specified error status value
printLogo() – print the VxWorks logo
logout() – log out of the VxWorks system
h() – display or set the size of shell history
spyReport() – display task activity data
spyTask() – run periodic task activity reports
spy() – begin periodic task activity reports
spyClkStart() – start collecting task activity data
spyClkStop() – stop collecting task activity data
spyStop() – stop spying and reporting
spyHelp() – display task monitoring help menu
unld() – unload an object module by specifying a file name or module ID (shell command)
reld() – reload an object module (shell command)

DESCRIPTION

This library consists of routines meant to be executed from the VxWorks shell. It provides useful utilities for task monitoring and execution, system information, symbol table management, etc.

Many of the routines here are simply command-oriented interfaces to more general routines contained elsewhere in VxWorks. Users should feel free to modify or extend this library, and may find it preferable to customize capabilities by creating a new private library, using this one as a model, and appropriately linking the new one into the system.

Some routines here have optional parameters. If those parameters are zero, which is what the shell supplies if no argument is typed, default values are typically assumed.

A number of the routines in this module take an optional task name or ID as an argument. If this argument is omitted or zero, the "current" task is used. The current task (or "default" task) is the last task referenced. The **usrLib** library uses **shellTaskIdDefault()** to set and get the last-referenced task ID of the current shell session. This routine calls in turn

taskIdDefault() to the system default task accordingly, in order that other VxWorks routines have a correct behavior.

Note that if a task name is provided but this name matches a symbol name (e.g. a routine), the order of precedence for the symbol resolution in the shell will prevent finding the task. In this case, use double quotes around the task name.

INCLUDE FILES	usrLib.h
SEE ALSO	usrFsLib , usrRtpLib , spyLib , dbgLib , the VxWorks programmer guides.

usrRtpLib

NAME	usrRtpLib – Real Time Process user interface subroutine library
ROUTINES	rtpHelp() – print a synopsis of RTP-related shell commands rtpLkup() – list symbols from an RTP's symbol table rtpLkAddr() – list symbols in an RTP whose values are near a specified value rtpSp() – launch a RTP with default options. rtpSymsAdd() – add symbols from an executable file to a RTP symbol table rtpSymsRemove() – remove symbols from a RTP symbol table rtpSymsOverride() – override the RTP symbol registration policy shlSymsAdd() – add symbols from a shared object file to a RTP symbol table shlSymsRemove() – remove shared library symbols from a RTP symbol table rtpi() – display all tasks within an RTP
DESCRIPTION	<p>This library consists of routines related to Real Time Processes (RTP) which are meant to be executed from the C interpreter of the VxWorks shell. It provides useful utilities for RTP monitoring and execution, system information, symbol table management, etc.</p> <p>Some routines here have optional parameters. If those parameters are zero, which is what the shell supplies if no argument is typed, default values are typically assumed.</p>
INCLUDE FILES	N/A
SEE ALSO	usrLib , rtpLib , windsh , the VxWorks programmer guides.

usrRtpStartup

NAME	usrRtpStartup – RTP Startup Facility Support Code
------	--

ROUTINES	startupScriptFieldSplit() – Split the startup script field of the bootline
DESCRIPTION	This module implements support code for the RTP Startup Facility. The function startupScriptFieldSplit splits the startup script field of the bootline at the first occurrence of a # character and null-terminates it at that location. The text before the # is the name of a traditional startup script file containing shell commands. Everything following the first # is part of a list of RTP's to startup.
INCLUDE FILES	none

usrShellHistLib

NAME	usrShellHistLib – shell history user interface subroutine library
ROUTINES	histSave() – save history of the current shell session interpreter(s) histLoad() – load history into the current shell session interpreter(s)
DESCRIPTION	This library contains user callable routines to save and load the shell history for the current shell session. This can be done for all interpreters or only for the current interpreter.
INCLUDE FILES	none

usrTransLib

NAME	usrTransLib – Transaction Device Access Library
ROUTINES	usrFormatTrans() – Perform a low-level trans XBD format operation usrTransCommit() – Set a transaction point on a trans XBD usrTransCommitFd() – set a transaction point using a file descriptor
DESCRIPTION	This module provides access to special functionality of the Transactional Extended Block Device (also known as the "trans XBD" and "TRFS"). It provides functions for both low-level formatting and setting transaction points.
INCLUDE FILES	transCbio.h, usrTransLib.h

utfLib

NAME	utfLib – Library to manage Unicode characters encoded in UTF-8 and UTF-16
ROUTINES	utfLibInit() – initialize the UTF library proofUtf8() – Determine if a string represents a valid UTF-8 character utf8ToCP() – Convert a UTF-8 encoded Unicode character to the Unicode codepoint. CPToUtf8() – Convert a Unicode codepoint to a UTF-8 encoding utf16ToCP() – Convert a UTF-16 encoded Unicode character to a codepoint. CPToUtf16() – Convert a Unicode codepoint to a UTF-16 encoding utfflen8() – return the encoding length of a NULL terminated UTF-8 string utfflen16() – Return the number of 16-bit words used by a UTF-16 encoding. proofUtf8String() – determine if a string is valid UTF-8 utf8ToUtf16String() – convert a UTF-8 string to a UTF-16 string utf16ToUtf8String() – Convert a UTF-16 string to a UTF-8 String utf8ToUtf16StringBOM() – Convert UTF-8 to UTF16 with a Byte Order Mark utf16ToUtf8StringBOM() – Convert UTF-16 to UTF-8 based on a Byte Order Mark
DESCRIPTION	<p>This library provides conversion routines for transforming Unicode characters encoded in UTF-8 and several variations of UTF-16 to the corresponding Unicode codepoint, as well as encoding Unicode codepoints as UTF-8 and UTF-16.</p> <p>There are two basic types of conversion routines supplied - routines that convert between UTF-8 or UTF-16 encodings and codepoints, and routines that convert between UTF-8 and UTF-16 encoded strings.</p> <p>UTF-8 strings have no associated byte-order representation, however, UTF-16 has an associated byte-order. This library can handle byte order which is either indicated by the caller, or, in the case of strings, use a preceding Byte Order Mark as described in "The Unicode Standard, v 4.0".</p>
INCLUDE FILES	utfLib.h

virtualDiskLib

NAME	virtualDiskLib – virtual disk driver library (vxSim)
ROUTINES	virtualDiskInit() – install the virtual disk driver virtualDiskCreate() – create a virtual disk device. virtualDiskClose() – close a virtual disk block device.

DESCRIPTION

The module provides APIs to emulate a VxWorks disk driver. The VxWorks disk appears under a single host file.

The host file name and the disk formatting parameters must be specified to `virtualDiskCreate()` when creating the virtual disk. The parameters are then stored into the virtual disk host file. If the virtual disk already exists, all information regarding structure of the disk (number of bytes per block, number of blocks per track and number of blocks on this device) are read from the existing virtual disk; values passed in parameter of the `virtualDiskCreate()` routine are ignored.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The routine `virtualDiskInit()` must be called to initialize the driver and the `virtualDiskCreate()` routine is used to create devices. Before removing the host file associated with the virtual disk, the routine `virtualDiskClose()` must be called.

CREATING VIRTUAL DISKS

Before a virtual disk can be used, it must be created. This is done with the `virtualDiskCreate()` call. The format of this call is:

```
BLK_DEV * virtualDiskCreate
(
  char * hostFile,           /* name of the host file to use      */
  int   bytesPerBlk,        /* number of bytes per block         */
  int   blksPerTrack,       /* number of blocks per track        */
  int   nBlocks             /* number of blocks on this device   */
)
```

The *hostFile* parameter specifies the name of the host file used for the virtual disk. The host file pathname is a standard host pathname without the host name. For Windows VxSim, the path separator to use is either `\` or `/` (i.e. `c:/myDir/myFile` or `c:\myDir\myFile`).

The *bytesPerBlk* parameter specifies the size of each logical block on the disk. If *bytesPerBlk* is zero, 512 is the default.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the host disk. If *blksPerTrack* is zero, the count of blocks per track will be set to *nBlocks* (i.e., the disk will be defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, 512 is the default.

The formatting parameters (*bytesPerBlk*, *blksPerTrack*, and *nBlocks*) are critical only if the host file does not exist. In that case, the formatting parameters are used to compute the disk size and to fill the block device structure.

The disk size is defined by `bytesPerBlk * nBlocks`. *blksPerTrack* is used to calculate the number of track.

The `virtualDiskCreate()` API is not able to open an old unix disk created with the `unixDiskDevCreate()` API.

The `virtualDiskCreate ()` call returns a pointer to a block device structure. This structure contains fields that describe the physical properties of a disk device and specify the addresses of routines within the virtual disk driver.

Once the device has been created, it still does not have a name or file system associated with it. This can be done by creating an XBD wrapper for the returned **BLK_DEV** pointer (eg. `xbdBlkDevCreate (pBlkDev, "/deviceName")`). The file system framework will attempt to identify and instantiate the type file system (if any) currently installed on the virtual disk. If no identifiable file system was found, then it will be instantiated with `rawFs`. The type of file system installed on the virtual disk can be changed at any time by formatting it with the desired file system (eg. **`dosFsVolFormat ()`**, **`hrfsFormat ()`**).

As an example, to create a 208KB disk, 512-byte blocks, and 32 blocks per track, the proper call would be:

```
BLK_DEV * pBlkDev;

pBlkDev = virtualDiskCreate ("c:/tmp/filesys1", 512, 32, 416);
```

This will attach the host file `c:/tmp/filesys1` as a block device.

CLOSING VIRTUAL DISKS

Closing a virtual disk is done with the **`virtualDiskClose ()`** call. The format of this call is:

```
STATUS virtualDiskClose
(
    BLK_DEV blkDev; /* virtual disk block device to close */
)
```

The *blkDev* parameter specifies the virtual disk block device to close.

The `virtualDiskClose ()` API close the host file associated with the virtual disk. The virtual disk referenced by `blkDev` can then be removed if needed.

IOCTL Only the `FIODISKFORMAT` request is supported; all other ioctl requests return an error, and set the task's `errno` to `S_ioLib_UNKNOWN_REQUEST`.

INCLUDE FILES **`virtualDiskLib.h`**

SEE ALSO **`xbdBlkDevCreate ()`**, **`dosFsVolFormat ()`**, **`hrfsFormat ()`**, **`dosfsDiskFormat ()`**, **`hrfsDiskFormat ()`**

vmArch32Lib

NAME **`vmArch32Lib`** – VM (VxVMI) library for PentiumPro/2/3/4 32 bit mode

ROUTINES **`vmArch32LibInit ()`** – initialize the arch specific unbundled VM library (VxVMI Option)

vmArch32Map() – map 32bit physical space into 32bit virtual space (VxVMI Option)
vmArch32Translate() – translate a 32bit virtual address to a 32bit physical address (VxVMI Option)

DESCRIPTION	<p>This library provides the virtual memory mapping and virtual address translation that works with the unbundled VM library VxVMI. The architecture specific VM library APIs are linked in automatically when INCLUDE_MMU_FULL and INCLUDE_MMU_P6_32BIT are both defined in the BSP. The provided APIs are vmArch32Map() and vmArch32Translate().</p> <p>The 4KB-page and 4MB-page are supported. The page size is configurable by VM_PAGE_SIZE macro in the BSP.</p>
INCLUDE FILES	mmuPro32Lib.h
SEE ALSO	vmLib , Intel Architecture Software Developer's Manual

vmArch36Lib

NAME	vmArch36Lib – VM (VxVMI) library for PentiumPro/2/3/4 36 bit mode
ROUTINES	<p>vmArch36LibInit() – initialize the arch specific unbundled VM library (VxVMI Option) vmArch36Map() – map 36bit physical space into 32bit virtual space (VxVMI Option) vmArch36Translate() – translate a 32bit virtual address to a 36bit physical address (VxVMI Option)</p>
DESCRIPTION	<p>The 36 bit physical addressing mechanism of P6 (Pentium II and III) family processors and P7 (Pentium4) family processors are supported by this library. This library provides the virtual memory mapping and virtual address translation that works with the unbundled VM library VxVMI. The architecture specific VM library APIs are linked in automatically when INCLUDE_MMU_FULL and INCLUDE_MMU_P6_36BIT are both defined in the BSP. The provided APIs are vmArch36Map() and vmArch36Translate().</p> <p>The 4KB-page and 2MB-page are supported. The page size is configurable by VM_PAGE_SIZE macro in the BSP.</p> <p>The memory description table sysPhysMemDesc[] in sysLib.c has not changed and only allows 32 bit virtual and physical address mapping description. Thus the first 4GB 32 bit address space of the 36 bit physical address is used at the initialization/boot time. Then, the physical address beyond the 4GB can be mapped in somewhere outside of the system memory pool in the 32 bit virtual address space with above APIs.</p>
INCLUDE FILES	mmuPro36Lib.h

SEE ALSO **vmLib**, Intel Architecture Software Developer's Manual

vmBaseArch32Lib

NAME	vmBaseArch32Lib – VM (bundled) library for PentiumPro/2/3/4 32 bit mode
ROUTINES	vmBaseArch32LibInit() – initialize the arch specific bundled VM library vmBaseArch32Map() – map 32bit physical to the 32bit virtual memory vmBaseArch32Translate() – translate a 32bit virtual address to a 32bit physical address
DESCRIPTION	<p>This library provides the virtual memory mapping and virtual address translation that works with the bundled VM library. The architecture specific VM library APIs are linked in automatically when INCLUDE_MMU_BASIC and INCLUDE_MMU_P6_32BIT are both defined in the BSP. The provided APIs are vmBaseArch32Map() and vmBaseArch32Translate().</p> <p>The 4KB-page and 4MB-page are supported. The page size is configurable by VM_PAGE_SIZE macro in the BSP.</p>
INCLUDE FILES	mmuPro32Lib.h
SEE ALSO	vmLib , Intel Architecture Software Developer's Manual

vmBaseArch36Lib

NAME	vmBaseArch36Lib – VM (bundled) library for PentiumPro/2/3/4 36 bit mode
ROUTINES	vmBaseArch36LibInit() – initialize the arch specific bundled VM library vmBaseArch36Map() – map 36bit physical to the 32bit virtual memory vmBaseArch36Translate() – translate a 32bit virtual address to a 36bit physical address
DESCRIPTION	<p>The 36 bit physical addressing mechanism of P6 (Pentium II and III) family processors and P7 (Pentium4) family processors are supported by this library. This library provides the virtual memory mapping and virtual address translation that works with the bundled VM library. The architecture specific VM library APIs are linked in automatically when INCLUDE_MMU_BASIC and INCLUDE_MMU_P6_36BIT are both defined in the BSP. The provided APIs are vmBaseArch36Map() and vmBaseArch36Translate().</p> <p>The 4KB-page and 2MB-page are supported. The page size is configurable by VM_PAGE_SIZE macro in the BSP.</p>

The memory description table `sysPhysMemDesc[]` in **sysLib.c** has not changed and only allows 32 bit virtual and physical address mapping description. Thus the first 4GB 32 bit address space of the 36 bit physical address is used at the initialization/boot time. Then, the physical address beyond the 4GB can be mapped in somewhere outside of the system memory pool in the 32 bit virtual address space with above APIs.

INCLUDE FILES **mmuPro36Lib.h**

SEE ALSO **vmLib**, Intel Architecture Software Developer's Manual

vmBaseLib

NAME **vmBaseLib** – base virtual memory support library

ROUTINES

- vmBaseStateSet()** – change the state of a block of virtual memory (obsolete)
- vmStateSet()** – change the state of a block of virtual memory
- vmStateGet()** – get the state of a page of virtual memory
- vmBasePageSizeGet()** – return the MMU page size (obsolete)
- vmPageSizeGet()** – return the page size
- vmTranslate()** – translate a virtual address to a physical address
- vmPhysTranslate()** – translate a physical address to a virtual address
- vmMap()** – map physical space into virtual space
- vmPageMap()** – map physical space into virtual space
- vmTextProtect()** – write-protect kernel text segment
- vmPageOptimize()** – Optimize the address range if possible.
- vmPageLock()** – lock the pages.
- vmPageUnlock()** – unlock the pages.

DESCRIPTION This library provides the MMU (Memory Management Unit) support needed for kernel facilities and kernel applications. To enable this feature, configure VxWorks with the **INCLUDE_MMU_BASIC** component. Note that **INCLUDE_MMU_GLOBAL_MAP** is also a required component.

It also provides support for the following features:

- Ability to write protect the kernel text segment as well as the text segments loaded by the incremental loader.
- Ability to get the state of a virtual page.
- Ability to translate a physical address to virtual address and vice versa.
- Ability to map a page.
- Ability to be able to write to a buffer regardless of the protection attributes.

- Enable certain processor specific optimizations, such as TLB locking and large pages. For details see the respective Architecture Supplement.

For code that must not rely on the VM to be initilized or included (so that dependancy with VM is not introduced), the VM routines should be accessed through the following macros. These macros first check that the routines they call are installed, which allows the use of a scaled back VM layer.

VM_STATE_SET(context, virtAdrs, len, stateMask, state)
This calls **vmStateSet()**.

VM_STATE_GET(context, pageAddr, pState)
This calls **vmStateGet()**.

VM_PAGE_MAP(context, virtualAddr, physicalAddr, len)
This calls **vmMap()**.

VM_PAGE_SIZE_GET()
The calls **vmPageSizeGet()**.

VM_TRANSLATE(context, virtualAddr, physicalAddr)
This calls **vmTranslate()**.

VM_PHYS_TRANSLATE(context, physicalAddr, virtualAddr)
This calls **vmPhysTranslate()**.

VM_CONTEXT_BUFFER_WRITE(context, fromAddr, toAddr, nbBytes)
This calls **vmBufferWrite()**.

This macro should be used only in very specific cases where the current context cannot be defined using **VM_CURRENT_GET()** and therefore the context to use must be passed to **vmBufferWrite()**. Nevertheless the context ID passed to **VM_CONTEXT_BUFFER_WRITE()** must correspond to the context in place.

VM_PAGE_OPTIMIZE(context, virtAddr, len, option)
This calls **vmPageOptimize()**.

VM_PAGE_LOCK(context, virtAddr, len, option)
This calls **vmPageLock()**.

VM_PAGE_UNLOCK(context, virtAddr)
This calls **vmPageUnlock()**.

INCLUDE FILES **sysLib.h, vmLib.h**

SEE ALSO **vmGlobalMap, vmShow**, the VxWorks programmer guides.

vmGlobalMap

NAME	vmGlobalMap – virtual memory global mapping library																													
ROUTINES	vmGlobalMapInit() – initialize global mapping vmBaseGlobalMapInit() – initialize global mapping (obsolete)																													
DESCRIPTION	<p>This library provides the minimal virtual memory functionality for initializing global MMU mappings for the kernel. These mappings are created at system startup based on the system memory descriptor table - sysPhysmemDesc[] - of the BSP. This functionality is enabled with the INCLUDE_VM_GLOBAL_MAP component.</p> <p>The mappings can be modified at runtime only if the vmBaseLib API is included in the system with the INCLUDE_MMU_BASIC component.</p> <p>The physical memory descriptor contains information used to initialize the MMU attribute information in the translation table. The following state bits may be or'ed together:</p> <p>Supervisor access:</p> <table><tr><td>MMU_ATTR_PROT_SUP_READ</td><td>read access in supervisor mode</td></tr><tr><td>MMU_ATTR_PROT_SUP_WRITE</td><td>write access in supervisor mode</td></tr><tr><td>MMU_ATTR_PROT_SUP_EXE</td><td>executable access in supervisor mode</td></tr></table> <p>User access:</p> <table><tr><td>MMU_ATTR_PROT_USR_READ</td><td>read access in user mode</td></tr><tr><td>MMU_ATTR_PROT_USR_WRITE</td><td>write access in user mode</td></tr><tr><td>MMU_ATTR_PROT_USR_EXE</td><td>executable access in user mode</td></tr></table> <p>Validity attribute:</p> <table><tr><td>MMU_ATTR_VALID</td><td>page is valid.</td></tr></table> <p>Cache attributes:</p> <table><tr><td>MMU_ATTR_CACHE_OFF</td><td>cache turned off</td></tr><tr><td>MMU_ATTR_CACHE_COPYBACK</td><td>cache in copy-back mode</td></tr><tr><td>MMU_ATTR_CACHE_WRITETHRU</td><td>cache set in writethrough mode</td></tr><tr><td>MMU_ATTR_CACHE_GUARDED</td><td>page access set to guarded.</td></tr><tr><td>MMU_ATTR_CACHE_COHERENCY</td><td>page access set to cache coherent.</td></tr><tr><td>MMU_ATTR_CACHE_DEFAULT</td><td>default cache value, set with USER_D_CACHE_MODE</td></tr></table> <p>Additionally, mask bits are or'ed together in the initialStateMask structure element to describe which state bits are being specified in the initialState structure element:</p> <table><tr><td>MMU_ATTR_PROT_MSK</td></tr><tr><td>MMU_ATTR_CACHE_MSK</td></tr></table>		MMU_ATTR_PROT_SUP_READ	read access in supervisor mode	MMU_ATTR_PROT_SUP_WRITE	write access in supervisor mode	MMU_ATTR_PROT_SUP_EXE	executable access in supervisor mode	MMU_ATTR_PROT_USR_READ	read access in user mode	MMU_ATTR_PROT_USR_WRITE	write access in user mode	MMU_ATTR_PROT_USR_EXE	executable access in user mode	MMU_ATTR_VALID	page is valid.	MMU_ATTR_CACHE_OFF	cache turned off	MMU_ATTR_CACHE_COPYBACK	cache in copy-back mode	MMU_ATTR_CACHE_WRITETHRU	cache set in writethrough mode	MMU_ATTR_CACHE_GUARDED	page access set to guarded.	MMU_ATTR_CACHE_COHERENCY	page access set to cache coherent.	MMU_ATTR_CACHE_DEFAULT	default cache value, set with USER_D_CACHE_MODE	MMU_ATTR_PROT_MSK	MMU_ATTR_CACHE_MSK
MMU_ATTR_PROT_SUP_READ	read access in supervisor mode																													
MMU_ATTR_PROT_SUP_WRITE	write access in supervisor mode																													
MMU_ATTR_PROT_SUP_EXE	executable access in supervisor mode																													
MMU_ATTR_PROT_USR_READ	read access in user mode																													
MMU_ATTR_PROT_USR_WRITE	write access in user mode																													
MMU_ATTR_PROT_USR_EXE	executable access in user mode																													
MMU_ATTR_VALID	page is valid.																													
MMU_ATTR_CACHE_OFF	cache turned off																													
MMU_ATTR_CACHE_COPYBACK	cache in copy-back mode																													
MMU_ATTR_CACHE_WRITETHRU	cache set in writethrough mode																													
MMU_ATTR_CACHE_GUARDED	page access set to guarded.																													
MMU_ATTR_CACHE_COHERENCY	page access set to cache coherent.																													
MMU_ATTR_CACHE_DEFAULT	default cache value, set with USER_D_CACHE_MODE																													
MMU_ATTR_PROT_MSK																														
MMU_ATTR_CACHE_MSK																														

MMU_ATTR_VALID_MSK

If there is an error when mappings are initialized, the reason for the failure is recorded in sysExcMsg area and the system is rebooted. The most common reasons for failures are the invalid combination of state/stateMask, or overlapping and conflicting entries in sysPhysemDec[].

INCLUDE FILES none

SEE ALSO **vmBaseLib**

vmShow

NAME **vmShow** – virtual memory show routines

ROUTINES **vmContextShow()** – display the translation table for a context
vmAttrShow() – display the text representation of a MMU attribute value

DESCRIPTION This library contains virtual memory information display routines.

The routine **vmShowInit()** links this facility into the VxWorks system. It is called automatically when this facility is configured into VxWorks.

CONFIGURATION To use the virtual memory show routines, configure VxWorks with the
INCLUDE_MMU_FULL_SHOW component.

AVAILABILITY This module is distributed as the bundled virtual memory support option.

INCLUDE FILES **vmLib.h**

vrfsLib

NAME **vrfsLib** – the Virtual Root File System

ROUTINES **vrfsInit()** – Initialize the Virtual Root File System Library
vrfsDevCreate() – Instantiate the VRFS

DESCRIPTION	This component provides a file system representing the contents of the Core IO device table, thus providing a root file system, the elements of which are all Core IO devices, whose names begin with "/" and do not contain an embedded "/".
INCLUDE FILES	none

vxAtomicLib

NAME	vxAtomicLib – atomic operations library
ROUTINES	<p> vxAtomicAdd() – atomically add a value to a memory location vxAtomicSub() – atomically subtract a value from a memory location vxAtomicInc() – atomically increment a memory location vxAtomicDec() – atomically decrement a memory location vxAtomicOr() – atomically perform a bitwise OR on memory location vxAtomicXor() – atomically perform a bitwise XOR on a memory location vxAtomicAnd() – atomically perform a bitwise AND on a memory location vxAtomicNand() – atomically perform a bitwise NAND on a memory location vxAtomicSet() – atomically set a memory location vxAtomicGet() – atomically get a memory location vxAtomicClear() – atomically clear a memory location vxCas() – atomically compare-and-swap the contents of a memory location VX_MEM_BARRIER_W() – Write memory barrier VX_MEM_BARRIER_R() – Read Memory Barrier VX_MEM_BARRIER_RW() – Read/Write Memory Barrier </p>
DESCRIPTION	This library provides routines to perform a number of atomic operations on a memory location: add, subtract, increment, decrement, bitwise OR, bitwise NOR, bitwise AND, bitwise NAND, set, clear and compare-and-swap. In addition, this library also provides memory barrier APIs that enforce memory access orders for CPU to CPU interaction.

ATOMIC OPERATORS

Atomic operations constitute one of the solutions to the mutual exclusion problems faced by multi-threaded applications. The ability to perform an indivisible read-modify-write operation on a memory location allows multiple threads of execution, tasks or ISRs, to safely read-modify-write a global variable. Mutex semaphores, interrupt locking and task locking are other mutual exclusion mechanisms that exist in VxWorks.

- Atomic operators can be used from both task and interrupt level, allowing a variable to safely be read-modified-written from any context. Using mutex semaphores to control access to a variable from interrupt level is not possible as these cannot be taken or given from interrupt level.

- Atomic operators leverage the capabilities of the hardware to perform atomic operations. Therefore they are very fast since little processing is required.
- Unlike task locking atomic operators can be used in a multiprocessing environment where multiple CPUs share a global memory location.
- Unlike interrupt locking, atomic operators can be used in a multiprocessing environment where multiple CPUs share a global memory location.

Atomic operations are performed on *atomic_t* type variables. Depending on the underlying processor architecture, restrictions may exist regarding the memory alignment and caching attributes of *atomic_t* variables. Below is a list of the instructions used to implement atomic operators. Refer to the relevant processor user's manual to determine the restrictions that apply to the use of these instructions.

PowerPC

lwarx/stwcx

Intel Architecture

lock/cmpxchg

MIPS

ll/sc

MEMORY BARRIERS

Memory barriers are ways to enforce an ordering between memory access operations on either side of the barriers. Barriers ensure that the sequence of memory operations performed by one CPU appears the same to the rest of the system, especially when there is a possibility of interaction between two CPUs. However, if other mechanisms are used, such as semaphores, to ensure that interactions between two CPUs are safe, then memory barriers are not necessary.

There are three types of memory barriers provided:

VX_MEM_BARRIER_W() - Write memory barrier

A write memory barrier guarantees that all store memory operations before the write barrier have occurred before any subsequent store operations after the write barrier.

VX_MEM_BARRIER_R() - Read Memory Barrier

A read memory barrier guarantees that all load memory operations before the read memory barrier have occurred before any subsequent load operations after the read barrier.

VX_MEM_BARRIER_RW() - Read/Write Memory Barrier

A Read/Write memory barrier is essentially a general barrier that enforces ordering for either reads or writes. Hence, a R/W memory barrier can be used in substitution for either a read or a write barrier.

In VxWorks SMP, there are a few locking strategies that enforce memory barriers. The following is a list of routines that enforce a full memory barrier.

- spinLockXXXTake() APIs and spinLockXXXGive() APIs
- semTake() and semGive() APIs

Other terms such as "membar" or memory fence are often used in software documentation. These terms are equivalent to memory barriers.

INCLUDE FILES vxAtomicLib.h

vxCpuLib

NAME	vxCpuLib – CPU utility routines
ROUTINES	<p>vxCpuEnabledGet() – get a set of running CPUs</p> <p>vxCpuConfiguredGet() – get the number of configured CPUs in the system</p> <p>vxCpuIndexGet() – get the index of the calling CPU</p>
DESCRIPTION	<p>This library provides a small number of utility routines for users who need to have visibility into the number of CPUs that are present in a VxWorks system. One typical example is code that needs to manage per-CPU objects, which are objects that need to be replicated for each CPU and accessed based on the CPU a task or ISR is running on.</p> <p>Routines in this library allow a user to determine:</p> <ul style="list-style-type: none">- The CPU on which the calling task or ISR is presently running.- The number of CPUs configured in the system.- The number of enabled CPUs in the system. <p>There are two important concepts that need to be understood by users of this library: CPU indices and CPU sets.</p>
CPU INDICES	<p>In a VxWorks single CPU system, the CPU index is always 0.</p> <p>In a VxWorks SMP system each CPU is uniquely identified using an index. The index is an unsigned integer ranging from 0 to N-1 where N is the number of CPUs configured in the system. The CPU index of the bootstrap CPU is always 0. This is the CPU that takes the system out of reset and enables other CPUs at boot time. The number of configured CPUs in a system is set at compile time.</p>
CPU SETS	<p>Routine vxCpuEnabledGet() returns the set of enabled CPUs in the system. In VxWorks a set of CPUs is always represented using a cpuset_t type variable. Refer to the reference entry for cpuset to obtain more information.</p>
INCLUDE FILES	vxCpuLib.h

SEE ALSO cpuset, The VxWorks Programmer's Guides

vxLib

NAME vxLib – miscellaneous support routines

ROUTINES vxTas() – C-callable atomic test-and-set primitive
 vxMemArchProbe() – architecture specific part of vxMemProbe
 vxMemProbe() – probe an address for a bus error
 vxSSEnable() – enable the superscalar dispatch (MC68060)
 vxSSDisable() – disable the superscalar dispatch (MC68060)
 vxPowerModeSet() – set the power management mode (PowerPC, SH, x86)
 vxPowerModeGet() – get the power management mode (PowerPC, SH, x86)
 vxPowerDown() – place the processor in reduced-power mode (PowerPC, SH)
 vxCr0Get() – get a content of the Control Register 0 (x86)
 vxCr0Set() – set a value to the Control Register 0 (x86)
 vxCr2Get() – get a content of the Control Register 2 (x86)
 vxCr2Set() – set a value to the Control Register 2 (x86)
 vxCr3Get() – get a content of the Control Register 3 (x86)
 vxCr3Set() – set a value to the Control Register 3 (x86)
 vxCr4Get() – get a content of the Control Register 4 (x86)
 vxCr4Set() – set a value to the Control Register 4 (x86)
 vxEflagsGet() – get a content of the EFLAGS register (x86)
 vxEflagsSet() – set a value to the EFLAGS register (x86)
 vxDrGet() – get a content of the Debug Register 0 to 7 (x86)
 vxDrSet() – set a value to the Debug Register 0 to 7 (x86)
 vxTssGet() – get a content of the TASK register (x86)
 vxTssSet() – set a value to the TASK register (x86)
 vxGdtrGet() – get a content of the Global Descriptor Table Register (x86)
 vxIdtrGet() – get a content of the Interrupt Descriptor Table Register (x86)
 vxLdtrGet() – get a content of the Local Descriptor Table Register (x86)

DESCRIPTION This module contains miscellaneous VxWorks support routines.

INCLUDE FILES vxLib.h

vxMemProbeLib

NAME vxMemProbeLib – miscellaneous support routines

ROUTINES	vxMemProbeInit() – add vxMemProbeTrap exception handler to exc handler chain vxMemProbe() – probe an address for a bus error
DESCRIPTION	This module provides an architecture-independent mechanism for supporting vxMemProbe.
INCLUDE FILES	none

vxbEtsecEnd

NAME	vxbEtsecEnd – Freescale Enhanced TSEC VxBus END driver
ROUTINES	etsecRegister() – register with the VxBus subsystem
DESCRIPTION	<p>This module implements a driver for the Motorola/Freescale Enhanced Three Speed Ethernet Controller (ETSEC) network interface. The ETSEC supports 10, 100 and 1000Mbps operation over copper and fiber media.</p> <p>The ETSEC is unusual in that it uses three different interrupt vectors: one for RX events, one for TX events and one for error events. The intention is to shave some cycles from the interrupt service path by jumping directly to the proper event handler routine instead of the driver having to determine the nature of pending events itself.</p> <p>Note that while this driver is VxBus-compliant, it does not use vxbDmaBufLib. The reason for this is that vxbDmaBufLib is technically only required for drivers for DMA-based devices that must be portable among multiple architectures (e.g. PCI or VMEbus adapters). The ETSEC is not a standalone device: it only exists as an integrated component of certain MPC85xx and MPC83xx PowerPC CPUs. It is always big-endian, it is always cache-coherent (since we always enable the ETSEC's snooping features), and it never needs bounce buffering or address translation. Given this, we may as well forgo the use of vxbDmaBufLib entirely, since using it will do nothing except add a bit of extra overhead to the packet processing paths.</p> <p>The ETSEC supports several advanced features not present in the original TSEC. This includes TCP/IP checksum offload support, hardware VLAN tag insertion and stripping, hardware packet parsing and filing, and transmit packet prioritization. The receive filter logic also supports a wider multicast hash filter (512 bits vs 256 in the original TSEC) and a 16 entry CAM filter. This driver includes support for the checksum features and the filter, and uses the CAM filter and expanded hash table to provide improved multicast filtering, but does not support the VLAN tag insertion/stripping feature or the transmit prioritization.</p> <p>The VLAN tagging/stripping is not supported because of an incompatibility between the hardware and our VLAN support code. When VLAN insertion is enabled for transmit, the</p>

hardware always inserts a VLAN tag; if no frame control block with a valid VLAN control field is present, it uses the value from the DFVLAN register. This is not the behavior we expect: the decision whether or not to insert a tag at all should be done on a per-frame basis, not globally. The ETSEC's design makes it impossible to send untagged frames as long as the VLINS bit in the transmit control register is set, which is not what we want. (I am mystified as to why Freescale felt the need to implement it this way.)

The transmit packet prioritization is not supported, because it requires the use of multiple transmit DMA queues. Currently, there is no API available in the MUX or the TCP/IP stack to enable us to support multiple TX queues.

For the RX frame parser and filer, the ETSEC supports up to 8 physical DMA queues, and up to 64 virtual queues. Currently, we support only 8 queues, with the queue ID corresponding to one of the physical DMA rings. When the filer is enabled, special device-specific ioctls can be used to program the filer to distribute frames among the queues depending on the various filtering properties supported by the hardware. A few simple cases are also provided as examples. Currently, the MUX/END API does not support multiple input queues, so the driver uses the filer and RX queues to prioritize the order in which the frames are sent to the stack. The driver considers queue 0 to have the highest priority and queue 7 the lowest. When frames arrive, those directed to queue 0 will be processed first, followed by queue 1, and queue 2, and so on. If the filer is programmed to reject certain frames, they won't be seen at all.

Since using the filer requires configuring all 8 RX DMA queues, which consumes a large amount of memory, the filer support is off by default. It can be enabled using the "filerEnable" configuration parameter, as shown below.

SMP CONSIDERATIONS

While the ETSEC has three interrupt vectors, it has only one interrupt status register and one interrupt mask register. Each interrupt service routine can only mask off specific interrupt events without touching the others (i.e. the RX ISR can only mask RX events). Doing this requires clearing bits in the mask register using a read-modify-write operation, which is not atomic. A similar read-modify-write operation is used in the task level interrupt handler code to un-mask interrupts as well. For proper SMP synchronization, these operations must be guarded with a spinlock.

AMP CONSIDERATIONS

The MPC8641D dual core processor contains 4 ETSEC devices, and can be run in either SMP or AMP mode. In AMP mode, it may be desirable to allocate the on-board ETSEC interface among the cores, i.e. assigning ETSEC0 and ETSEC1 to core0, and ETSEC2 and ETSEC3 to core1. There are two complications involved in doing this.

The first is that while there are four complete ETSEC controllers, there is only one functional MDIO port (namely the one associated with ETSEC0). Reading or writing a PHY register through the MDIO registers is not an atomic operation, which means simultaneous accesses by both cores will overlap and likely fail. To guard against this, the tsecMdio driver should be used in conjunction with the ETSEC driver. This driver provides a simple inter-core

synchronization mechanism which allows both cores to share access to the MDIO port without contention.

The second complication involves address translation. To use VxWorks in AMP mode on the 8641D, low memory offset mode must be enabled. In this mode, the second core applies a bias of 256MB to memory addresses. The Freescale 8641D eval board is configured with 512MB of RAM: using low memory offset mode effectively splits the memory into 256MB banks, with core0 using the first bank and core1 using the second. Using low memory offset mode allows the second core to think that the second bank of RAM starting at address 0x10000000 actually starts at address 0x0. This allows the same VxWorks image linked for core0 to run on core1 unmodified.

This bias only applies to the CPU core, however: it does not apply to the ETSEC controllers, which still need to be provided with the absolute physical addresses of descriptors and packet buffers in order to perform DMA correctly. This means that when running on the second core, the ETSEC driver must perform a virtual to physical address translation on all DMA addresses. A *physMask* configuration property is provided to specify the virtual to physical bias offset. For ETSEC instances allocated to core0, this property can be left undefined, or explicitly set to 0. For ETSEC instances allocated to core1, *physMask* should be set to 0x10000000.

Assigning an ETSEC instance to a particular core is done by setting the *coreNum* configuration property. If *coreNum* is not defined, or explicitly set to 0 for a particular ETSEC instance, then that instance will be assigned to core0. If *coreNum* is set to 1, it will be assigned to core1 instead. Any ETSEC can be assigned to any core (the BSP defaults to assigning ETSEC0 and ETSEC1 to the first core and ETSEC2 and ETSEC3 to the second).

BOARD LAYOUT The ETSEC is directly integrated into the CPU. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides a vxBus external interface. The only exported routine is the **etsecRegister()** function, which registers the driver with VxBus. Since the ETSEC is a processor local bus device, each device instance must be specified in the **hwconf.c** file in a BSP. The hwconf entry must specify the following parameters:

regBase

Specifies the base address where the controller's CSR registers are mapped into the host's address space. All register offsets are computed relative to this address.

intr0

Specifies the interrupt vector for the ETSEC's TX interrupts.

intrLevel0

Specifies the interrupt level for the ETSEC's TX interrupts.

intr1

Specifies the interrupt vector for the ETSEC's RX interrupts.

intrLevel1

Specifies the interrupt level for the ETSEC's RX interrupts.

intr2

Specifies the interrupt vector for the ETSEC's error interrupts.

intrLevel2

Specifies the interrupt level for the ETSEC's error interrupts.

tbiAddr

Specifies the address to use for the internal TBI management interface. This value will be programmed into the TBIPA register, and controls the address at which the TBI management registers will be visible when accessing the MII management port. When the ETSEC is not operating in TBI mode, this value should be set so that it does not conflict with the address of a copper PHY, or else the management registers of the copper PHY will be obscured.

phyAddr

Specifies the address of the PHY allocated to this ETSEC instance. On most boards, all PHYs share the same management port, so we must specify explicitly which PHY on the management bus maps to which ETSEC.

miilfName

Specifies the name of the VxBus device driver that manages this ETSEC's MDIO management port. In the UP and SMP cases, motetsec0 is designated as the manager of the MDIO port. For AMP, this must be set to the "tsecMdio" driver: this driver has special support for synchronizing access to ETSEC0's MDIO registers between cores.

miilfUnit

The unit number that goes with *miilfName*. Together, these describe a specific VxBus instance (i.e. motetsec0, or tsecMdio0).

coreNum

Specifies the core to which to allocate this particular ETSEC in an AMP configuration. If this property is omitted or explicitly set to 0, then the ETSEC instance will be bound to core0. If set to 1, it will be bound to core1. Any ETSEC may be mapped to any core. In UP and SMP mode, this property should not be used.

physMask

Specifies the memory offset to apply when translating virtual addresses to physical addresses on ETSECs that are bound to the second core in an AMP configuration. For ETSECs bound to core0, this property should be ommitted or set to 0. For ETSECs bound to core1, it should be set to 0x10000000. This property should only be used when *coreNum* is set to 1.

An example hwconf entry is shown below:

```
const struct hcfResource tsecVxbEnd0Resources[] = {
    { "regBase",          HCF_RES_INT,    { (void *) (CCSBAR + 0x24000) } },
    { "intr0",            HCF_RES_INT,    { (void *) EPIC_TSEC1TX_INT_VEC } },
    { "intr0Level",       HCF_RES_INT,    { (void *) EPIC_TSEC1TX_INT_VEC } },
```



```

    { "intr1",          HCF_RES_INT,    { (void *)EPIC_TSEC1RX_INT_VEC } },
    { "intr1Level",    HCF_RES_INT,    { (void *)EPIC_TSEC1RX_INT_VEC } },
    { "intr2",          HCF_RES_INT,    { (void *)EPIC_TSEC1ERR_INT_VEC } },
    { "intr2Level",    HCF_RES_INT,    { (void *)EPIC_TSEC1ERR_INT_VEC } },
    { "phyAddr",        HCF_RES_INT,    { (void *)0 } },
#ifdef INCLUDE_AMP
    { "miiIfName",      HCF_RES_STRING, { (void *)"tsecMdio" } },
    { "miiIfUnit",      HCF_RES_INT,    { (void *)0 } },
    { "coreNum",        HCF_RES_INT,    { (void *)0 } }
#else
    { "miiIfName",      HCF_RES_STRING, { (void *)"motetsec" } },
    { "miiIfUnit",      HCF_RES_INT,    { (void *)0 } }
#endif
};
#define tsecVxbEnd0Num NELEMENTS(tsecVxbEnd0Resources)

```

The ETSEC controller also supports jumbo frames. This driver has jumbo frame support, which is disabled by default in order to conserve memory (jumbo frames require the use of an buffer pool with larger clusters). Jumbo frames can be enabled on a per-interface basis using a parameter override entry in the **hwconf.c** file in the BSP. For example, to enable jumbo frame support for interface motetsec0, the following entry should be added to the **VXB_INST_PARAM_OVERRIDE** table:

```
{ "motetsec", 0, "jumboEnable", VXB_PARAM_INT32, {(void *)1} }
```

The ETSEC controller also supports interrupt coalescing. This driver has coalescing support, which is disabled by default so that the **out of the box** configuration has the smallest interrupt latency. Coalescing can be enabled on a per-interface basis using parameter overrides in the **hwconf.c** file, in the same way as jumbo frame support. In addition to turning the coalescing support on and off, the timeout and packet count values can be set:

```

{ "motetsec", 0, "coalesceEnable", VXB_PARAM_INT32, {(void *)1} }
{ "motetsec", 0, "coalesceRxTicks", VXB_PARAM_INT32, {(void *)10} }
{ "motetsec", 0, "coalesceRxPkts", VXB_PARAM_INT32, {(void *)8} }
{ "motetsec", 0, "coalesceTxTicks", VXB_PARAM_INT32, {(void *)100} }
{ "motetsec", 0, "coalesceTxPkts", VXB_PARAM_INT32, {(void *)16} }

```

If only the *coalesceEnable* property is set, the driver will use default timeout and packet count values as shown above. Specifying alternate values via the BSP will override the defaults.

To enable support for the filer and multiple RX queues, the *filerEnable* property should also be set, as illustrated below:

```
{ "motetsec", 0, "filerEnable", VXB_PARAM_INT32, {(void *)1} }
```

Note that the filer support can not be enabled or disabled on the fly at runtime.

INCLUDE FILES none

SEE ALSO vxBus, **ifLib**, miiBus, "Writing an Enhanced Network Driver", "MPC8548E PowerQUICC III Integrated Communications Processor Reference Manual",
http://www.freescale.com/files/32bit/doc/ref_manual/MPC8548ERM.pdf

vxbFileNvRam

NAME	vxbFileNvRam – VxBus driver for NVRam on a filesystem file
ROUTINES	vxbFileNvRamRegister() – register vxbFileNvRam driver vxbFileNvRamDrvCtrlShow() – show pDrvCtrl for template controller vxbFileNvRamGet() – get the contents of non-volatile RAM vxbFileNvRamSet() – write to non-volatile RAM
DESCRIPTION	This is the VxBus driver for the VXB_FILERAM device for use by battery backed RAM devices and other byte-oriented non-volatile RAM, which can be read and written as if it were normal RAM.
INCLUDE FILES	none

vxbI8042Kbd

NAME	vxbI8042Kbd – Intel 8042 keyboard driver routines
ROUTINES	i8042vxbRegister() – register i8042vxb driver
DESCRIPTION	This is the driver for the Intel 8042 Keyboard Controller Chip used on a personal computer 386 / 486. This driver handles the standard 101 key board. This driver does not change the defaults set by the BIOS. The BIOS initializes the scan code set to 1 which make the PS/2 keyboard compatabile with the PC and PC XT keyboard.

USER CALLABLE ROUTINES

Some routines in this driver are accessed via VxBus methods.

The **i8042Intr()** is the interrupt handler which handles the key board interrupt and is responsible for the handing the character received to whichever console the device is initialized to. By default this is initialized to **PC_CONSOLE**. If the user has to change the current console he will have to make an ioctl call with the option **CONIOCURCONSOLE** and the argument as the console number which he wants to change to. To return to the

console owned by the shell the user has to do an ioctl call back to the console number owned the shell from his application.

NOTES

A `hcfResource` structure must be defined in the BSP's `hwconf.c` file which sets the values for `regBase` (the I/O address of the data register for the keyboard), `irq` (interrupt vector), `regInterval` (set to 4), `irqLevel` (interrupt level), and `mode` (0 for Japanese, 1 for English). This is one example:

```
const struct hcfResource i8042KbdResources[] =
{
    { "regBase",    HCF_RES_INT, {(void *)DATA_8042} },
    { "irq",        HCF_RES_INT, {(void *) (INUM_TO_IVEC(INT_NUM_KBD))} },
    { "regInterval", HCF_RES_INT, {(void *)4} },
    { "irqLevel",   HCF_RES_INT, {(void *)KBD_INT_LVL} },
    { "mode",       HCF_RES_INT, {(void *)KEYBRD_MODE} }
};
```

The macros `N_VIRTUAL_CONSOLES` and `PC_CONSOLE` should be defined in `config.h` file.

INCLUDE FILES

none

SEE ALSO

`vxblPcConsole.c`

vxblntellchStorage

NAME

`vxblntellchStorage` – Intel ICH0/1 (82801) ATA/IDE and ATAPI CDROM

ROUTINES

`vxblntellchStorageRegister()` – register driver with `vxbus`
`ichAtaDrv()` – Initialize the ATA driver
`ichAtaXbdDevCreate()` – create an XBD device for a ATA/IDE disk
`ichAtaDevCreate()` – create a device for a ATA/IDE disk
`ichAtaBlkRW()` – read or write sectors to a ATA/IDE disk.
`ichAtapiPktCmdSend()` – Issue a Packet command.
`ichAtapiIoctl()` – Control the drive.
`atapiParamsPrint()` – Print the drive parameters.
`ichAtapiCtrlMediumRemoval()` – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command
`ichAtapiRead10()` – read one or more blocks from an ATAPI Device.
`ichAtapiReadCapacity()` – issue a READ CD-ROM CAPACITY command to a ATAPI device
`ichAtapiReadTocPmaAtip()` – issue a READ TOC command to a ATAPI device
`ichAtapiScan()` – issue SCAN packet command to ATAPI drive.

ichAtapiSeek() – issues a SEEK packet command to drive.
ichAtapiSetCDSpeed() – issue SET CD SPEED packet command to ATAPI drive.
ichAtapiStopPlayScan() – issue STOP PLAY/SCAN packet command to ATAPI drive.
ichAtapiStartStopUnit() – Issues START STOP UNIT packet command
ichAtapiTestUnitRdy() – issue a TEST UNIT READY command to a ATAPI drive
ichAtaCmd() – issue a RegisterFile command to ATA/ATAPI device.
ichAtaInit() – initialize ATA device.
ichAtaRW() – read/write a data from/to required sector.
ichAtaDmaRW() – read/write a number of sectors on the current track in DMA mode
ichAtaPiInit() – init a ATAPI CD-ROM disk controller
ichAtaDevIdentify() – identify device
ichAtaParamRead() – Read drive parameters
ichAtaCtrlReset() – reset the specified ATA/IDE disk controller
ichAtaStatusChk() – Check status of drive and compare to requested status.
ichAtapiPktCmd() – execute an ATAPI command with error processing
ichAtapiInit() – init ATAPI CD-ROM disk controller
ichAtaXbdRawio() – do raw I/O access
ichAtaRawio() – do raw I/O access
ichAtaConfig() – configure an ATA drive (hard disk or cdrom drive)
ichAtaConfigInit() – initialize the hard disk driver

DESCRIPTION

BLOCK DEVICE DRIVER:

This is a Block Device Driver for ATA/ATAPI devices on IDE host controller. It also provides necessary functions to user for device and its features control which are not used or utilized by file system.

This driver provides standard Block Device Driver functions,(blkRd, blkWrt, ioctl, statusChk, and reset) for ATA and ATAPI devices separately as the scheme of implementation differs. These functions are implemented as **ataBlkRd()**, **ataBlkWrt()**, **ataBlkIoctl()**, **ataStatus()** and **ataReset()** for ATA devices and **atapiBlkRd()**, **atapiBlkWrt()**, **atapiBlkIoctl()**, **atapiStatusChk()** and **atapiReset()** for ATAPI devices. The Block Device Structure **BLK_DEV** is updated with these function pointers at initialization of the driver depending on the type of the device in function **ichAtaDevCreate()**.

ichAtaDrv(), a user callable function, initializes ATA/ATAPI devices present on the specified IDE controller(either primary or secondary), which must be called once for each controller, before usage of this driver, usually called from **usrRoot()** in **usrConfig.c**.

The routine **ichAtaDevCreate()**, which is user callable function, is used to mount a logical drive on an ATAPI drive. This routine returns a pointer to **BLK_DEV** structure, which is used to mount the file system on the logical drive.

OTHER NECESSARY FUNCTIONS FOR USER:

There are various functions provided to user, which can be classified to different categories as device control function, device information functions and functions meant for packet devices.

Device Control Function:

ichAtapiIoctl() function is used to control a device. Block Device Driver functions **ataBlkIoctl()** and **atapiBlkIoctl()** functions are also routed to this function. This function implements various control command functions which are not used by the I/O system (like power management feature set commands, host protected feature set commands, security feature set commands, media control functions etc).

Device Information Function:

In this category various functions are implemented depending on the information required. These functions return information required (like cylinder count, Head count, device serial number, device Type, etc) from the internal device structures.

Packet Command Functions:

Although Block Device Driver functions deliver packet commands using functions provided by **atapiLib.c** for required functionality. There are group of functions provided in this driver to user for ATAPI device, which implements packet commands for **CD_ROM** that comply to **ATAPI-SFF8020i** specification which are essentially required for CD ROM operation for file system. These functions are named after their command name (like for REQUEST SENSE packet command **atapiReqSense()** function). To issue other packet commands **ichAtapiPktCmdSend()** can be used.

This driver also provides a generic function **ichAtapiPktCmdSend()** to issue a packet command to ATAPI devices, which can be utilized by user to issue packet command directly instead using the implemented functions also may be used to send new commands (may come in later specs) to device. User can issue any packet command using **ichAtapiPktCmdSend()** function to the required device by passing its **BLK_DEV** structure pointer and pointer for **ATAPI_CMD** command packet.

typedef of **ATAPI_CMD**

```
typedef struct atapi_cmd
{
    UINT8          cmdPkt [ATAPI_MAX_CMD_LENGTH];
    char           **ppBuf;
    UINT32         bufLength;
    ATA_DATA_DIR   direction;
    UINT32         desiredTransferSize;
    BOOL           dma;
    BOOL           overlap;
} ATAPI_CMD;
```

and **ATA_DATA_DIR** typedef is

```
typedef enum /* with respect to host/memory */
{
```

```
NON_DATA, /* non data command */
OUT_DATA, /* to drive from memory */
IN_DATA /* from drive to memory */
} ATA_DATA_DIR;
```

User is expected supposed to fill the **ATAPI_CMD** structure with required parameters of the packet and pass the **ATAPI_CMD** structure pointer to **ichAtapiPktCmdSend()** fuunction for command execution.

All the packet command functions require **ATA_DEV** structure to be passed, which alternatively a **BLK_DEV** Device Structure of the device. One should type convert the structure and the same **BLK_DEV** structrue pointer to these functions.

The routine **ataPiRawio()** supports physical I/O access. The first argument is the controller number, 0 or 1; the second argument is drive number, 0 or 1; the third argument is a pointer to an **ATA_RAW** structure.

PARAMETERS:

The **ataPiDrv()** function requires a configuration flag as a parameter. The configuration flag is one of the following or Bitwise OR of any of the following combination:

configuration flag =

Transfer mode | Transfer bits | Transfer unit | Geometry parameters

Transfer mode	Description	Transfer Rate
ATA_PIO_DEF_0	PIO default mode	
ATA_PIO_DEF_1	PIO default mode, no IORDY	
ATA_PIO_0	PIO mode 0	3.3 MBps
ATA_PIO_1	PIO mode 1	5.2 MBps
ATA_PIO_2	PIO mode 2	8.3 MBps
ATA_PIO_3	PIO mode 3	11.1 MBps
ATA_PIO_4	PIO mode 4	16.6 MBps
ATA_PIO_AUTO	PIO max supported mode	
ATA_DMA_SINGLE_0	Single DMA mode 0	2.1 MBps
ATA_DMA_SINGLE_1	Single DMA mode 1	4.2 MBps
ATA_DMA_SINGLE_2	Single DMA mode 2	8.3 MBps
ATA_DMA_MULTI_0	Multi word DMA mode 0	4.2 MBps
ATA_DMA_MULTI_1	Multi word DMA mode 1	13.3 MBps
ATA_DMA_MULTI_2	Multi word DMA mode 2	16.6 MBps
ATA_DMA_ULTRA_0	Ultra DMA mode 0	16.6 MBps
ATA_DMA_ULTRA_1	Ultra DMA mode 1	25.0 MBps
ATA_DMA_ULTRA_2	Ultra DMA mode 2	33.3 MBps
ATA_DMA_ULTRA_3	Ultra DMA mode 3	44.4 MBps
ATA_DMA_ULTRA_4	Ultra DMA mode 4	66.6 MBps
ATA_DMA_ULTRA_5	Ultra DMA mode 5	100.0 MBps
ATA_DMA_AUTO	DMA max supported mode	
Transfer bits		

ATA_BITS_16	RW bits size, 16 bits
ATA_BITS_32	RW bits size, 32 bits
Transfer unit	
ATA_PIO_SINGLE	RW PIO single sector
ATA_PIO_MULTI	RW PIO multi sector
Geometry parameters	
ATA_GEO_FORCE	set geometry in the table
ATA_GEO_PHYSICAL	set physical geometry
ATA_GEO_CURRENT	set current geometry

ISA SingleWord DMA mode is obsolete in ata-3.

The Transfer rates shown above are the Burst transfer rates. If **ATA_PIO_AUTO** is specified, the driver automatically chooses the maximum PIO mode supported by the device. If **ATA_DMA_AUTO** is specified, the driver automatically chooses the maximum Ultra DMA mode supported by the device and if the device doesn't support the Ultra DMA mode of data transfer, the driver chooses the best Multi Word DMA mode. If the device doesn't support the multiword DMA mode, driver chooses the best single word DMA mode. If the device doesn't support DMA mode, driver automatically chooses the best PIO mode. So it is recommended to specify the **ATA_DMA_AUTO**.

If **ATA_PIO_MULTI** is specified, and the device does not support it, the driver automatically chooses single sector or word mode. If **ATA_BITS_32** is specified, the driver uses 32-bit transfer mode regardless of the capability of the drive. The Single word DMA mode will not be supported by the devices compliant to ATA/ATAPI-5 or higher.

This driver supports UDMA mode data transfer from device to host, provided 80 conductor cable is used for required controller device. This check done at the initilisation of the device from the device parameters and if 80 conductor cable is connected then UDMA mode transfer is selected for operation subject to condition that required UDMA mode is supported by device as well as host. This driver follows ref-3 Chapter 4 "Determining a Drive's Transfer Rate Capability" to determine drives best transfer rate for all modes (ie UDMA, MDMA, SDMA and PIO modes).

The host IDE Bus master functions are to be mapped to follwing macro defined for various functionality in header file which are used in this driver.

ATA_HOST_CTRL_INIT - initialize the controller
ATA_HOST_DMA_ENGINE_INIT - initialize bus master DMA engine
ATA_HOST_DMA_ENGINE_START - Start bus master operation
ATA_HOST_DMA_ENGINE_STOP - Stop bus master operation
ATA_HOST_DMA_TRANSFER_CHK - check bus master data transfer complete
ATA_HOST_DMA_MODE_NEGOTIATE - get mode supported by controller
ATA_HOST_SET_DMA_RWMODE - set controller to required mode
ATA_HOST_CTRL_RESET - reset the controller

If **ATA_GEO_PHYSICAL** is specified, the driver uses the physical geometry parameters stored in the drive. If **ATA_GEO_CURRENT** is specified, the driver uses current geometry parameters initialized by BIOS. If **ATA_GEO_FORCE** is specified, the driver uses geometry parameters stored in **sysLib.c**.

The geometry parameters are stored in the structure table **ataTypes[]** in **sysLib.c**. That table has two entries, the first for drive 0, the second for drive 1. The members of the structure are:

```
int cylinders;           /* number of cylinders */
int heads;               /* number of heads */
int sectors;             /* number of sectors per track */
int bytes;               /* number of bytes per sector */
int precomp;             /* precompensation cylinder */
```

The driver supports two controllers and two drives on each. This is dependent on the configuration parameters supplied to **ataPiDrv()**.

SMP CONSIDERATIONS

Most of the processing in this driver occurs in the context of a dedicated task, and therefore is inherently SMP-safe. One area of possible concurrence occurs in the interrupt service routine, **ataIntr()**. An ISR-callable spin lock take/give pair has been placed around the code which acknowledges/clears the ATA controller's interrupt status register. If the BSP or application provides functions for **ataIntPreProcessing** or **ataIntPostProcessing**, consideration will have to be given to making these functions SMP-safe. Most likely, some portion(s) of these functions will need to be protected by a spin lock. The spin lock allocated for the controller can be used. Consult the SMP Migration Guide for hints.

References:

- 1) ATAPI-5 specification "T13-1321D Revision 1b, 7 July 1999"
- 2) ATAPI for CD-ROMs "SFF-8020i Revision 2.6, Jan 22, 1996"
- 3) Intel 82801BA (ICH2), 82801AA (ICH), and 82801AB (ICH0) IDE Controller Programmer's Reference Manual, Revision 1.0 July 2000

Source of Reference Documents:

- 1) <ftp://ftp.t13.org/project/d1321r1b.pdf>
- 2) <http://www.bswd.com/sff8020i.pdf>

INCLUDE FILES none

SEE ALSO *VxWorks Programmer's Guide: I/O System*

vxblIntelIchStorageShow

NAME	vxblIntelIchStorageShow – ICH ATA disk device driver show routine
ROUTINES	<p>ichAtaShowInit() – initialize the ATA/IDE disk driver show routine</p> <p>ichAtaShow() – show the ATA/IDE disk parameters</p> <p>ichAtaDmaToggle() – turn on or off an individual controllers dma support</p> <p>ichAtapiCylinderCountGet() – get the number of cylinders in the drive.</p> <p>ichAtapiHeadCountGet() – get the number heads in the drive.</p> <p>ichAtapiDriveSerialNumberGet() – get the drive serial number.</p> <p>ichAtapiFirmwareRevisionGet() – get the firm ware revision of the drive.</p> <p>ichAtapiModelNumberGet() – get the model number of the drive.</p> <p>ichAtapiFeatureSupportedGet() – get the features supported by the drive.</p> <p>ichAtapiFeatureEnabledGet() – get the enabled features.</p> <p>ichAtapiMaxUDmaModeGet() – get the Maximum Ultra DMA mode the drive can support.</p> <p>ichAtapiCurrentUDmaModeGet() – get the enabled Ultra DMA mode.</p> <p>ichAtapiMaxMDmaModeGet() – get the Maximum Multi word DMA mode the drive supports.</p> <p>ichAtapiCurrentMDmaModeGet() – get the enabled Multi word DMA mode.</p> <p>ichAtapiMaxSDmaModeGet() – get the Maximum Single word DMA mode the drive supports</p> <p>ichAtapiCurrentSDmaModeGet() – get the enabled Single word DMA mode.</p> <p>ichAtapiMaxPioModeGet() – get the Maximum PIO mode that drive can support.</p> <p>ichAtapiCurrentPioModeGet() – get the enabled PIO mode.</p> <p>ichAtapiCurrentRwModeGet() – get the current Data transfer mode.</p> <p>ichAtapiDriveTypeGet() – get the drive type.</p> <p>ichAtapiVersionNumberGet() – get the ATA/ATAPI version number of the drive.</p> <p>ichAtapiRemovMediaStatusNotifyVerGet() – get the Media Stat Notification Version.</p> <p>ichAtapiCurrentCylinderCountGet() – get logical number of cylinders in the drive.</p> <p>ichAtapiCurrentHeadCountGet() – get the number of read/write heads in the drive.</p> <p>ichAtapiBytesPerTrackGet() – get the number of Bytes per track.</p> <p>ichAtapiBytesPerSectorGet() – get the number of Bytes per sector.</p> <p>ichAtaDumptest() – a quick test of the dump functionality for ATA driver</p>
DESCRIPTION	This library contains a driver show routine for the ATA/IDE (PCMCIA and LOCAL) devices supported on the IBM PC.
INCLUDE FILES	none

vxbM6845Vga

NAME	vxbM6845Vga – motorola 6845 VGA console driver
ROUTINES	m6845vxbRegister() – register m6845vxb driver
DESCRIPTION	This is the driver fo Video Contoroller Chip (6845) normally used in the 386/486 personal computers.

USER CALLABLE ROUTINES

This driver provides several VxBus methods for external access to its routines.

All virtual consoles are mapped to the same screen buffer. This is a very basic implementation of virtual consoles. Multiple screen buffers are not used to switch between consoles. This implementation is left for the future. Mutual exclusion for the screen buffer is guaranteed within the same console but it is not implemented across multiple virtual consoles because all virtual consoles use the same screen buffer. If multiple screen buffers are implemented then the mutual exclusion between virtual consoles can be implemented.

NOTES	The macro N_VIRTUAL_CONSOLES should be defined in config.h file. This refers to the number of virtual consoles which the user wishes to have. The user should define INCLUDE_ANSI_ESC_SEQUENCE in this file if the ansi escape sequences are required. Special processing in the m6845 driver is done if an escape sequence exists.
--------------	--

INCLUDE FILES	none
----------------------	------

SEE ALSO	tyLib , vxbPcConsole
-----------------	------------------------------------

vxbNonVolLib

NAME	vxbNonVolLib – non-volatile RAM to non-volatile memory routine mapping
ROUTINES	vxbNonVolLibInit() – Non Volatile RAM library initialization vxbNonVolGet() – get the contents of non-volatile RAM vxbNonVolSet() – write to non-volatile memory nvRamSegDefGet() – get segment allocation from BSP sysNetMacNVRamAddrGet() – get network MAC address from NVRAM
DESCRIPTION	This library provides the external API to handle non-volatile RAM manipulation in a VxBus environment.

INCLUDE FILES vxBus.h

vxbPcConsole

NAME **vxbPcConsole** – console handler

ROUTINES **pcConDrv()** – initialize the console driver
 pcConDevCreate() – create a device for the on-board ports
 pcConDevBind() – bind keyboard or VGA device with console

DESCRIPTION This file is used to link the keyboard driver and the vga driver.

USER CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, must be called directly: **pcConDrv()** to initialize the driver, and **pcConDevCreate()** to create devices.

Before using the driver, it must be initialized by calling **pcConDrv()** This routine should be called exactly once, before any reads, writes, or calls to **pcConDevCreate()**. Normally, it is called from **usrRoot()** in **usrConfig.c**.

Before a console can be used, it must be created using **pcConDevCreate()**.

IOCTL FUNCTIONS

This driver responds to the same ioctl codes as a normal ty driver.

NOTES The macro **N_VIRTUAL_CONSOLES** should be defined in **config.h** file.

INCLUDE FILES none

SEE ALSO **tyLib**

vxbSI31xxStorage

NAME **vxbSI31xxStorage** – PCI bus header file for vxBus

ROUTINES **vxbSI31xxStorageRegister()** – register driver with vxbus
 sil31xxDrvVxbInit() – Initialize the driver.
 sil31xxDiskPresent() – Return OK if disk exists.

sil31xxXbdCreate() – Create an XBD for the specified port.
sil31xxXbdDelete() – Delete an XBD for a specified port
sil31xxIsr() – Interrupt service routine.
sil31xxBIST() – Controller Built-In Self Test...
sil31xxBISTShow() – Show the results of the power-on BIST
sil31xxRegisterPortCallback() – register the port call back for a PHYRdyChg
sil31xxSectorRW() – read a single sector

DESCRIPTION

XBD driver for:

- Silicon Image 3132 SATA PCI Express controller
- Silicon Image 3124 SATA PCI-X controller

The 31xx driver can support either 2 or 4 ports. The pci device ID identifies the part as either a 3124 or a 3132. The 3124 supports 4 ports and the 3132 support 2 ports.

SIL31XX_MAX_CTRL is a limit on the number of chips (pci boards with this chip) that we can be configured in the system.

As each controller (chipset) is initialized, the pci device id is examined. Based on the type 3124 or 3132, the maximum number of ports for that device is saved in the control block. A port monitor task is spawned for each port.

This monitor task is blocked on the "deviceChange" semaphore. When it is recognized that a device has been inserted (via an ISR) the "deviceChange" semaphore is given to the port monitor task.

If the device change indicates that a device is present, the XBD interface to the device is created. 2 semaphores are used in this interface. A mutex to force mutual exclusion between the driver and the other users of the xbd interface, and a bio ready semaphore. Both of these semaphores are used by the "bio task" which is created at this time for each device (port) as the device becomes active. The bio ready indicates that there is work for the driver to do. Part of the creation of the xbd interface is to send an insertion event to the event reporting framework. This allows the filesystem to use the newly created xbd.

If the device change indicates that a device has gone away, the XBD interface is deleted, the bio service task is deleted and other resources used by the bio service task are recovered. The device change indication allow for hot removal/insertion of sata devices.

The interrupt handler has the controller specific control block passed as an argument by the pci interrupt routine. The isr scans each port on that controller for status bits. If the a device change has occurred, the "device change" is given to the waiting port monitor task. If the command has completed, the command complete semaphore is given. This semaphore serializes access to the driver. The bio task will call the function to read/write to the device, and uses this semaphore for that purpose.

Multiple instances:

There can be up to SIL31XX_MAX_CTRL instances of the driver. The driver will assign a controller number (0 thru (SIL31XX_MAX_CTRL -1)) as a means of identifying the

controller instance. If disks aren't named, they can always be referenced by ctrl_num, port_num for the purpose of testing and diagnostics. Also, the default names of the drives will be formed from the assigned controller number and port number. If port3 on controller 1 is present, the default device name is "/s1p3"

- XBD driver for:
- Silicon Image 3132 SATA PCI Express controller
 - Silicon Image 3124 SATA PCI-X controller

Initialization of the the driver and block devices are handled by the vxBus interface. The driver is fully initialized in vxBus "connection" phase.

INCLUDE FILES none

vxbsmscLan9118End

NAME vxbsmscLan9118End – SMSC LAN9118 VxBus END driver

ROUTINES smeRegister() – register with the VxBus subsystem

DESCRIPTION This module implements a driver for the SMSC LAN9118 single chip non-PCI 10/100 ethernet controller. The LAN9118 is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. The controller has an embedded 10/100 PHY, with simplified management interface.

The LAN9118 is a programmed I/O device: packet data transfer is driven by the host CPU, using separate 32 bit RX and TX FIFO I/O ports. The chip has 16K of internal FIFO memory which can be divided by the host between the RX and TX functions. Unlike the 91C111, the LAN9118 has a fully mapped register space, so that no switching between register banks is necessary.

The LAN9118 supports several different RX filtering modes, including hash filtering for either multicast or unicast. This driver configures the chip for "hash perfect" mode, using the single perfect filter for the station address and the 64 bit hash table for multicast traffic. The chip also has explicit support for all-multicast mode.

The internal 10/100 PHY is managed using miiBus, allowing fully dynamic link handling. The integrated PHY interrupt is used to provide instantaneous link change sensing.

The driver splits the 16K FIFO memory so that 6656 bytes are available for transmission, and the rest for reception. This allows up to 4 full size frames to be queued for transmission at any given time. Note that this is still quite small compared to other devices that support

DMA, so to avoid excessive TX frame drops, the protocol should implement some form of output queuing to buffer a reasonable amount of frames.

The device indicates RX and TX completion using RX and TX status FIFOs. When a frame reception completes, the chip writes a status word to one of the FIFOs. The device can be programmed to trigger an interrupt when a configurable amount of status words have been written to a status FIFO. To minimize latency, this driver always triggers an interrupt whenever any status words are available in one of the FIFOs.

BOARD LAYOUT The SMSC LAN9118 is typically wired directly to the host CPU. All configurations are jumperless.

EXTERNAL INTERFACE

The driver provides the standard VxBus external interface, **smeRegister()**. This function registers the driver with the VxBus subsystem, and instances will be created as needed. Since the LAN9118 is a processor local bus device, each device instance must be specified in the **hwconf.c** file in a BSP. The hwconf entry must specify the following parameters:

regBase

Specifies the base address where the controller's CSR registers are mapped into the host's address space. All register offsets are computed relative to this address.

intr

Specifies the interrupt vector for the LAN9118.

intrLevel

Specifies the interrupt level for the LAN9118.

An example hwconf entry is shown below:

```
struct hcfResource sme0Resources[] = {
    { "regBase",    HCF_RES_INT, {(void *)VERSATILE_SMC_ENET_BASE} },
    { "intr",       HCF_RES_INT, {(void *)INT_VEC_ETHERNET}},
    { "intrLevel",  HCF_RES_INT, {(void *)INT_LVL_ETHERNET}},
};
#define sme0Num NELEMENTS(sme0Resources)
```

A configuration parameter is also provided to set the default speed of the interface to 10Mbps instead of 100Mbps. Since this driver uses the LAN9118 in programmed I/O mode, performance at 100Mbps can be poor if no output queueing is provided by the stack. Setting the speed to 10Mbps improves the behavior somewhat. Forcing the interface to 10Mbps can be done by adding the following entry to the **VXB_INST_PARAM_OVERRIDE** table in the BSP's **hwconf.c** file:

```
{ "sme", 0, "lowSpeed", VXB_PARAM_INT32, {(void *)1} }
```

INCLUDE FILES none

SEE ALSO vxBus, ifLib, miiBus, "Writing an Enhanced Network Driver", "SMSC LAN9118 datasheet, <http://www.smssc.com/main/datasheets/9118.pdf>"

vxsimHostArchLib

NAME	vxsimHostArchLib – VxSim host side interface library
ROUTINES	vxsimHostDllLoad() – load the given DLL to VxSim. vxsimHostProcAddrGet() – return the address of a host API vxsimHostProcCall() – call a host routine vxsimHostMmuProtect() – set/clear protection on mmu pages vxsimHostMmuCurrentSet() – set current translation table mapping vxsimHostSioWrite() – write buffer to SIO device vxsimHostSioRead() – read SIO device into buffer vxsimHostSioOpen() – open SIO device vxsimHostSioClose() – close SIO device vxsimHostSioIntVecGet() – get SIO device interrupt vector vxsimHostSioModeSet() – set SIO device mode (poll/interrupt) vxsimHostSioBaudRateSet() – set SIO device transfert rate vxsimHostCpuVarsInit() – intialize per cpu variable pointers
DESCRIPTION	This module provide the ability to load and use dynamically loaded libraries (DLLs) from VxSim. This facility allows a VxSim application to reference any code located in a system or user DLL. This mechanism is very simple, the vxsimHostDllLoad() routine allows to load the DLL, and then vxsimHostProcAddrGet() can be used to retrieve the addresses of the DLL's exported routines.
IMPORTANT NOTE	Before invoking any host side routines, the interrupts must be locked to avoid system calls interruption.
EXAMPLE	The following code shows how to use these APIs: <pre>#include "vxWorks.h" #include "vxsimHostLib.h" STATUS dllTestStart (void) { FUNCPTR pDllTestInit; /* Load the test DLL */ if (vxsimHostDllLoad ("dllTest") == ERROR) { printf ("Error: Unable to load dllTest\n"); return (ERROR); } /* Get the address of the DLL init routine */ pDllTestInit = vxsimHostProcAddrGet ("testInit");</pre>

```

if (dllTestInit == NULL)
{
    printf ("Error: Unable to find testInit() symbol in loaded DLLs\n");
    return (ERROR);
}

/* invoke the DLL init routine */

vxsimHostProcCall (pDllTestInit, 0,0,0,0,0,0,0,0,0);

return (OK);
}

```

INCLUDE FILES **vxsimHostLib.h**

SEE ALSO **vxsim**

wdLib

NAME **wdLib** – watchdog timer library

ROUTINES **wdInitialize()** – initialize a pre-allocated watchdog.
wdStart() – start a watchdog timer
wdCancel() – cancel a currently counting watchdog
wdCreate() – create a watchdog timer
wdDelete() – delete a watchdog timer

DESCRIPTION This library provides a general watchdog timer facility. Any task may create a watchdog timer and use it to run a specified routine in the context of the system-clock ISR, after a specified delay.

Once a timer has been created with **wdCreate()**, it can be started with **wdStart()**. The **wdStart()** routine specifies what routine to run, a parameter for that routine, and the amount of time (in ticks) before the routine is to be called. (The timeout value is in ticks as determined by the system clock; see **sysClkRateSet()** for more information.) After the specified delay ticks have elapsed (unless **wdCancel()** is called first to cancel the timer) the timeout routine is invoked with the parameter specified in the **wdStart()** call. The timeout routine is invoked whether the task which started the watchdog is running, suspended, or deleted.

The timeout routine executes only once per **wdStart()** invocation; there is no need to cancel a timer with **wdCancel()** after it has expired, or in the expiration callback itself.

Note that the timeout routine is invoked at interrupt level, rather than in the context of the task. Thus, there are restrictions on what the routine may do. Watchdog routines are

constrained to the same rules as interrupt service routines. For example, they may not take semaphores, issue other calls that may block, or use I/O system routines like **printf()**.

Note: watchdog routine invocation can be deferred. As such `isrIdCurrent` is either a valid `ISR_ID` or is `NULL` in the case of deferral.

EXAMPLE In the fragment below, if **maybeSlowRoutine()** takes more than 60 ticks, **logMsg()** will be called with the string as a parameter, causing the message to be printed on the console. Normally, of course, more significant corrective action would be taken.

```
WDOG_ID wid = wdCreate ();
wdStart (wid, 60, logMsg, "Help, I've timed out!");
maybeSlowRoutine ();      /* user-supplied routine */
wdCancel (wid);
```

SMP CONSIDERATIONS Some or all of the APIs in this module are spinlock and `intCpuLock` restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are `intCpuLock` restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **wdLib.h**

SEE ALSO **logLib**, the VxWorks programmer guides.

wdShow

NAME **wdShow** – watchdog show routines

ROUTINES **wdShowInit()** – initialize the watchdog show facility
wdShow() – show information about a watchdog

DESCRIPTION This library provides routines to show watchdog statistics, such as watchdog activity, a watchdog routine, etc.

The routine **wdShowInit()** links the watchdog show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

- If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

- If you use the Tornado project facility, select **INCLUDE_WATCHDOGS_SHOW**.

SMP CONSIDERATIONS

Some or all of the APIs in this module are spinlock and intCpuLock restricted. Spinlock restricted APIs are the ones where it is an error condition for the caller to acquire any spinlock and then attempt to call these APIs. APIs that are intCpuLock restricted are the ones where it is an error condition for the caller to have disabled interrupts on the local CPU (by calling **intCpuLock()**) and then attempt to call these APIs. The method by which these error conditions are flagged and the exact behaviour in these situations are described in the individual API documentation.

INCLUDE FILES **wdLib.h**

SEE ALSO **wdLib**, **windsh**, the VxWorks programmer guides.

wdbLib

NAME **wdbLib** – WDB agent context management library

ROUTINES **wdbSystemSuspend()** – suspend the system

DESCRIPTION This library provides a routine to transfer control from the run time system to the WDB agent running in external mode. This agent in external mode allows a system-wide control, including ISR debugging, from a host tool (eg: Debugger, WindSh ...) through the target server and the WDB communication link.

ADDING WDB SUPPORT

To add WDB support, the component **INCLUDE_WDB** must be added to the kernel at configuration time. For more information about how to configure WDB see *Tornado User's Reference: Target Configuration*

INCLUDE FILES **wdb/wdbLib.h**

SEE ALSO

wdbMdlSymSyncLib

NAME **wdbMdlSymSyncLib** – target-host modules and symbols synchronization

ROUTINES **wdbMdlSymSyncLibInit()** – initialize modules and symbols synchronization library

DESCRIPTION This module provides host/target modules and symbols synchronization. With synchronization enabled, every module or symbol added to the run-time system from either the target or host side can be seen by facilities on both the target and the host. This synchronization makes it possible to use host tools to debug application modules loaded by the target loader.

The module is initialized by **wdbMdlSymSyncLibInit()**, which is called automatically when the WDB modules and symbols synchronization component (**INCLUDE_WDB_MDL_SYM_SYNC**) is included at configuration time.

When the target server connects the target agent, target and host symbol tables and module lists are synchronized so that every module loaded on the target before the target server was started can be seen by the host tools. This feature is particularly useful if VxWorks is started with a target-based startup script before the target server has been launched.

The target loader does not wait for synchronization completion to return. If your target server was started with the **-V** option, it prints a message indicating synchronization has been completed.

The following are examples of messages displayed by the target server indicating synchronization is complete:

```
Added module target_module      to target server... done
Added module host_module        to target... done
Added symbol targetSymbol       to target server... done
Added symbol hostSymbol         to target... done
```

If synchronization fails, the following message is displayed:

```
Added module host_module        to target... failed
```

This error generally means that synchronization of the corresponding module or symbol is not possible because there is not enough memory on the target.

Failure can also occur if a target server to target communication timeout is reached. If this is the case, you should stop the target server and restart it with the option **"-Bt"** to modify the WDB timeout between the target agent and the target server.

When a module is loaded from a host tool, the module is kept on the target when the target server disconnect from the target agent. If a new target server is started, this module is synchronized again and is visible from the new target server.

Note that the synchronization mechanism will not work when the WDB agent is running in system mode. The modules and symbols added while the agent in system mode will be synchronized later when a new module or symbol will be added while the agent is in task mode. However, this may lead to some unpredictable results.

INCLUDE FILES **wdb/wdbLib.h**

SEE ALSO **tgtsvr**

wdbUserEvtLib

NAME	wdbUserEvtLib – WDB user event library
ROUTINES	wdbUserEvtLibInit() – include the WDB user event library wdbUserEvtPost() – post a user event string to host tools
DESCRIPTION	<p>This library contains routines for sending WDB User Events. The event is sent through the WDB agent, the WDB communication link and the target server to the host tools that have registered for it. The event received by host tools will be a WTX user event string.</p> <p>To add WDB user events support, the component INCLUDE_WDB_USER_EVENT must be added at configuration time.</p>
INCLUDE FILES	wdb/wdbLib.h
SEE ALSO	

windPwrLib

NAME	windPwrLib – Power Management Library
ROUTINES	windPwrModeSet() – Set the BSP power mode windPwrModeGet() – Get the current power mode windPwrDownRtnSet() – register a BSP power-down function windPwrUpRtnSet() – register a BSP power-up function
DESCRIPTION	<p>This file provides methods for setting the processor into sleep mode and later waking it up. Power management operates in three states as defined by windPowerMode.</p> <p>windPwrModeOff In this CPU mode of operation, power management is disabled.</p> <p>windPwrModeShort In this CPU mode of operation, the CPU sleeps between system clock ticks but always wakes up for every tick. In other words, the tick interrupt source remains consistent off when this CPU power mode is set.</p> <p>windPwrModeLong This power mode is deprecated.</p> <p>The number of ticks that the BSP is requested to disable the tick interrupt source is provided by the kernel as a parameter to the BSP routine specified in windPwrDownRtnSet() as</p>

dRtn. When the CPU power down mode is windPwrModeLong, this value is used by dRtn to determine when to schedule an interrupt to wake up the processor. When the CPU power down mode is windPwrModeOff, dRtn is not invoked when the kernel goes idle. Hardware limitations may prevent the system from sleeping for the requested duration. If unable to support the requested duration, the BSP will program a sleep period for the maximum duration that hardware permits.

INCLUDE FILES **windPwrLib.h**

SEE ALSO the VxWorks programmer guides.

wvFileUploadPathLib

NAME **wvFileUploadPathLib** – file destination for event data

ROUTINES **wvFileUploadPathLibInit()** – initialize the **wvFileUploadPathLib** library
wvFileUploadPathCreate() – create a file for depositing event data
fileUploadPathClose() – close the event-destination file
wvFileUploadPathWrite() – write to the event-destination file

DESCRIPTION This file contains routines that write events to a file rather than uploading them to the host using a type of socket connection. If the file indicated is a TSFS file, this routine has the same result as uploading to a host file using other methods, allowing it to replace evtRecv. The file can be created anywhere, however, and event data can be kept on the target if desired.

INCLUDE FILES

SEE ALSO **wvSockUploadPathLib**, **wvTsfsUploadPathLib**

wvLib

NAME **wvLib** – event logging control library (System Viewer)

ROUTINES **wvLibInit()** – initialize **wvLib** - first step
wvLibInit2() – initialize **wvLib** - final step
wvEvtLogStart() – start logging events to the buffer
wvEvtLogStop() – stop logging events to the buffer
wvEvtClassSet() – set the class of events to log
wvEvtClassGet() – get the current set of classes being logged

wwEvtClassClear() – clear the specified class of events from those being logged
wwEvtClassClearAll() – clear all classes of events from those logged
wwObjInstModeSet() – set object instrumentation on/off
wwObjInst() – instrument objects
wwAllObjsSet() – set instrumented state for all objects and classes
wwSigInst() – instrument signals
wwSalInst() – instrument SAL
wwEventInst() – instrument VxWorks Events
wwEdrInst() – instrument ED&R Events
wwEvent() – log a user-defined event
wwUploadStart() – start upload of events to the host
wwUploadStop() – stop upload of events to host
wwUploadTaskConfig() – set priority and stacksize of **tWVUpload** task
wwLogCreate() – Create a System Viewer log
wwLogDelete() – Delete a System Viewer log
wwPartitionGet() – determine partition in use for System Viewer logging
wwPartitionSet() – specify a partition for use by System Viewer logging
wwLogListCreate() – create a list to hold System Viewer logs
wwLogListDelete() – delete a System Viewer log list
wwCurrentLogGet() – return a pointer to the currently active System Viewer log
wwCurrentLogSet() – select a System Viewer log as currently active
wwCurrentLogListSet() – set the current log list
wwCurrentLogListGet() – return a pointer to the System Viewer log list
wwLogFirstGet() – return a pointer to the first log in the System Viewer log list
wwLogNextGet() – return a pointer to the next log in the System Viewer log list
wwLogCountGet() – return the number of logs in the current log list

DESCRIPTION

This library contains routines that control event collection and upload of event data from the target to various destinations. The routines define the interface for the target component of the Wind River System Viewer. When event data has been collected, the routines in this library are used to produce event logs that can be loaded by the System Viewer host tools.

An event log is made up of a small header, a hash table which contains various important items, and a variable-sized binary data stream. The binary data carries the bulk of the events produced by the various event points throughout the kernel and associated libraries.

For convenience, the various parts of a System Viewer log are kept together in a **WV_LOG** structure, and a list of these structures is maintained in a **WV_LOG_LIST**.

In general, this information is gathered and stored temporarily on the target, and later uploaded to the host in the proper order to form an event log. The routines in this file can be used to create logs in various ways, depending on which routines are called, and in which order the routines are called.

There are three methods for uploading event logs. The first is to defer upload of event data until after logging has been stopped in order to eliminate events associated with upload activity from the event log. The second is to continuously upload event data as it is

gathered. This allows the collection of very large event logs, that may contain more events than the target event buffer can store at one time. The third is to defer upload of the data until after a target reboot.

It is possible to configure the buffer to allow old data to be overwritten by newer data. This allows logging to be carried out for arbitrarily long periods, and stopped when some fault condition is detected.

Each of these three methods is explained in more detail in CREATING AN EVENT LOG.

EVENT BUFFERS AND UPLOAD PATHS

Many of the routines in **wvLib** require access to the buffer used to store event data (the event buffer) and to the communication paths from the target to the host (the upload paths). Both the buffer and the path are referenced with IDs that provide **wvLib** with the appropriate information for access.

The event buffering mechanism used by **wvLib** is provided by **rBuffLib**. The upload paths available for use with **wvLib** are provided by **wvFileUploadPathLib**, **wvTsfsUploadPathLib** and **wvSockUploadPathLib**.

The upload mechanism backs off and retries writing to the upload path if an error occurs during the write attempt with the errno **EAGAIN** or **EWOULDBLOCK**. Two global variables are used to set the amount of time to back off and the number of retries. The variables are:

```
int wvUploadMaxAttempts    /* number of attempts to try writing */
int wvUploadRetryBackoff  /* delay between tries (in ticks - 60/sec) */
```

INITIALIZATION

This library is initialized in two steps. The first step, done by calling **wvLibInit()**, associates event logging routines to system objects. This is done when the kernel is initialized. The second step, done by calling **wvLibInit2()**, associates all other event logging routines with the appropriate event points. Initialization is done automatically when **INCLUDE_WINDVIEW** is defined.

DETERMINING WHICH EVENTS ARE COLLECTED

There are three classes of events that can be collected. They are:

```
WV_CLASS_1                /* Events causing context switches */
WV_CLASS_2                /* Events causing task-state transitions */
WV_CLASS_3                /* Events from object and system libraries */
```

The second class includes all of the events contained within the first class, plus additional events causing task-state transitions but not causing context switches. The third class contains all of the second, and allows logging of events within system libraries. It can also be limited to specific objects or groups of objects:

- Using **wvObjInst()** allows classes of objects (e.g. message queues) or individual instances (for example, **sem1**) to be instrumented.
- Using **wvSigInst()** allows signals to be instrumented.

- Using **wvEventInst()** allows vxWorks events (from **eventLib**) to be instrumented.
- Using **wvSalInst()** allows socket application layer (SAL) to be instrumented.

Logging events in Class 3 generates the most data, which may be helpful during analysis of the log. It is also the most intrusive on the system, and may affect timing and performance. Class 2 is more intrusive than Class 1. In general, it is best to use the lowest class that still provides the required level of detail.

To manipulate the class of events being logged, the following routines can be used: **wvEvtClassSet()**, **wvEvtClassGet()**, **wvEvtClassClear()**, and **wvEvtClassClearAll()**. To log a user-defined event, **wvEvent()** can be used. It is also possible to log an event from any point during execution using **e()**, located in **dbgLib**.

CONTROLLING EVENT LOGGING

Once the class of events has been specified, event logging can be started with **wvEvtLogStart()** and stopped with **wvEvtLogStop()**.

CREATING AN EVENT LOG

An event log consists of a number of components, grouped together in the **WV_LOG** structure. As discussed above, there are three common ways to upload an event log.

"Deferred Upload"

When creating an event log by uploading the event data after event logging has been stopped (deferred upload), the following series of calls can be used to start and stop the collection. In this example, the System Viewer log list and logs are created in the system memory partition. The event buffer should be allocated from the system memory partition as well. Error checking has been eliminated to simplify the example.

```
/* wvLib and rBuffLib initialized at system start up */

#include <vxWorks.h>
#include <wvLib.h>
#include <private/wvBufferP.h>
#include <private/wvUploadPathP.h>
#include <private/wvFileUploadPathLibP.h>
#include <sysLib.h>
#include <logLib.h>
#include <fcntl.h>

BUFFER_ID          bufId;
UPLOAD_ID          pathId;
WV_UPLOADTASK_ID  upTaskId;

/*
 * To prepare the event log and start logging:
 */

/* Select the system memory partition */

wvPartitionSet (memSysPartId);
```



```

/* If there is no log list, make one */

if (wvCurrentLogListGet () == NULL)
    wvLogListCreate ();

/* Create rBufs using default configuration */

bufId = rBuffCreate (&wvDefaultRBuffParams);

/* Create a System Viewer log, adding to the log list */

wvLogCreate (bufId);

wvEvtClassSet (WV_CLASS_1);          /* set to log class 1 events */
wvEvtLogStart ();

/*
 * To stop logging and complete the event log.
 */

wvEvtLogStop ();

/* Create an upload path using wvFileUploadPathLib, yielding pathId. */

pathId = wvFileUploadPathCreate ("/tgtsvr/eventLog.wvr", O_CREAT);

/* Start uploading, upload task will die when done */

upTaskId = wvUploadStart (wvCurrentLogGet (), pathId, FALSE);

/* Finish uploading */

wvUploadStop (upTaskId);

/* Close the upload path and destroy the event buffer */

wvFileUploadPathClose (pathId);

wvLogDelete (wvCurrentLogListGet (), wvCurrentLogGet ());

```

Routines which can be used as they are, or modified to meet the users' needs, are located in **usrWindview.c**. These routines, **wvOn()** and **wvOff()**, provide a way to produce useful event logs without using the host user interface of System Viewer.

"Continuous Upload"

When uploading event data as it is still being logged to the event buffer (continuous upload), simply rearrange the above calls:

```

/* Includes and declarations. */

BUFFER_ID          bufId;
UPLOAD_ID          pathId;
WV_UPLOADTASK_ID  upTaskId;

```

```
/*
 * To prepare the event log and start logging:
 */

/* Select the system memory partition */
wvPartitionSet (memSysPartId);

/* If there is no log list, make one */
if (wvCurrentLogListGet () == NULL)
    wvLogListCreate ();

/* Create rBufs using default configuration */
bufId = rBuffCreate (&wvDefaultRBuffParams);

/* Create a System Viewer log, adding to the log list */
wvLogCreate (bufId);

/* Create an upload path using wvFileUploadPathLib, yielding pathId. */
pathId = wvFileUploadPathCreate ("/tgtsvr/eventLog.wvr", O_CREAT);

/* Start uploading, upload task will wait */
upTaskId = wvUploadStart (wvCurrentLogGet (), pathId, TRUE);

wvEvtClassSet (WV_CLASS_1);          /* set to log class 1 events */
wvEvtLogStart ();

/* .... record system activity */
/*
 * To stop logging and complete the event log.
 */

wvEvtLogStop ();

wvUploadStop (upTaskId);

/* Close the upload path */
wvFileUploadPathClose (pathId);

wvLogDelete (wvCurrentLogListGet (), wvCurrentLogGet ());
```

"Post-Mortem Event Collection"

This library also contains routines that preserve task name information throughout event logging in order to produce post-mortem event logs.

Post-mortem event logs typically contain events leading up to a target failure. The memory containing the information to be stored in the log must not be zeroed when the system

reboots. The event buffer is set up to allow event data to be logged to it continuously, overwriting the data collected earlier. When event logging is stopped, either by a system failure or at the request of the user, the event buffer may not contain the first events logged due to the overwriting. As tasks are created the `EVENT_TASKNAME` that is used by the System Viewer host tools to associate a task ID with a task name can be overwritten, while other events pertaining to that task ID may still be present in the event buffer. In order to assure that the System Viewer host tools can assign a task name to a context, a copy of all task name events can be preserved outside the event buffer and uploaded separately from the event buffer.

Note that the `WV_LOG_LIST` structure contains a partition id. This allows the list to be created in a user-specified partition. For post-mortem data collection, the memory partition should be within memory that is not zeroed upon system reboot. The event buffer, preserved task names, and log header will be stored in the same partition.

Generating a post-mortem event log is similar to generating a deferred upload log. Typically event logging is stopped due to a system failure, but it may be stopped in any way. To retrieve the logs, the location of the log list in the preserved memory must be known.

```

/* Includes, as in the examples above. */

BUFFER_ID          bufId;
PART_ID            preservedPartition;
WV_LOG_LIST *      pWvLogList;

/*
 * To prepare the event log and start logging:
 */

pWvLogList = wvCurrentLogListGet ();

if (pWvLogList != NULL)
    wvLogListDelete (pWvLogList);

if (wvCurrentLogListGet () != NULL)
    wvLogListDelete ();

/*
 * Create a memory partition in the user-reserved region. It is assumed
 * that the whole of the region is available for System Viewer: This may
 * not be true, particularly if pmLib is included.
 * The first few words are not included in the partition, but are assumed
 * to be available. Conventionally, the pointer to the log list, which
 * contains all the required information, is stored at the start of the
 * user-reserved region, by default.
 */

preservedPartition = memPartCreate (sysMemTop () + 16,
                                   sysPhysMemTop () - sysMemTop () -
16);
if (preservedPartition == NULL)
{

```

```
        logMsg ("Error creating partition: Start 0x%x, Size: 0x%x\n",
        sysMemTop (), sysPhysMemTop () - sysMemTop (), 0, 0, 0, 0);
        return (ERROR);
    }

    wvPartitionSet (preservedPartition);

    pWvLogList = wvLogListCreate ();

    /*
    * Save the log list pointer in user-reserved memory
    */

    *(WV_LOG_LIST **)sysMemTop () = pWvLogList;

    /*
    * Create event buffer in non-zeroed memory, allowing overwrite,
    * yielding bufId. Set the wvDefaultRBufParams.option flags for this
    */

    wvDefaultRBufParams.sourcePartition = preservedMemPartition;
    rBuffId = rBuffCreate (&wvDefaultRBufParams);
    wvLogCreate (rBuffId);

    wvEvtClassSet (WV_CLASS_1);          /* set to log class 1 events */
    wvEvtLogStart ();

    /*
    * System fails and reboots. Note that the address of the WV_LOG_LIST
    * must be preserved through the reboot so the list can be set here
    * and used to upload the data. We assume that the pointer to the
    * WV_LOG_LIST, saved at the start of user-reserved memory, is
    * available.
    */
    */
```

After the target has rebooted, the log should be available in the user-reserved region

```
/* Includes, as in the examples above. */

UPLOAD_ID          pathId;
WV_UPLOADTASK_ID   upTaskId;
PART_ID            preservedPartition;
WV_LOG_LIST *      pWvLogList;

pWvLogList = *(WV_LOG_LIST **)sysMemTop ();

wvCurrentLogListSet (pWvLogList);

/* Create an upload path, yielding pathId. */

pathId = wvFileUploadPathCreate ("/tgtsvr/eventLog.wvr", O_CREAT);

upTaskId = wvUploadStart (wvLogFirstGet (), pathId, FALSE);
wvUploadStop (upTaskId);
```

```
/* Close the upload path and destroy the event buffer */  
  
wvFileUploadPathClose (pathId);  
  
/*  
 * Logs should not be deleted if the target has rebooted. This situation  
 * could be detected with a global variable which is set when the  
partition  
 * is created  
 * */  
  
/* wvLogDelete (wvCurrentLogListGet (), wvCurrentLogGet ()); */
```

INCLUDE FILES **wvLib.h eventP.h**

SEE ALSO **rBuffLib, wvFileUploadPathLib, wvSockUploadPathLib, wvTsfsUploadPathLib, Wind River System Viewer User's Guide**

wvSockUploadPathLib

NAME **wvSockUploadPathLib** – socket upload path library

ROUTINES **wvSockUploadPathLibInit()** – initialize **wvSockUploadPathLib** library
wvSockUploadPathCreate() – establish an upload path to the host using a socket
wvSockUploadPathClose() – close the socket upload path
wvSockUploadPathWrite() – write to the socket upload path

DESCRIPTION This file contains routines that are used by **wvLib** to pass event data from the target buffers to the host. This particular event-upload path opens a normal network socket connected with the Wind River System Viewer host process to transfer the data.

INCLUDE FILES

SEE ALSO **wvTsfsUploadPathLib, wvFileUploadPathLib**

wvTmrLib

NAME **wvTmrLib** – timer library (System Viewer)

ROUTINES	wvTmrRegister() – register a timestamp timer traceTmrResolutionGet() – get resolution of timestamp source, in nanoseconds
DESCRIPTION	<p>This library allows a Wind River System Viewer timestamp timer to be registered. When this timer is enabled, events are tagged with a timestamp as they are logged.</p> <p>Seven routines are required for System Viewer: a timestamp routine, a timestamp routine that guarantees interrupt lockout, a routine that enables the timer driver, a routine that disables the timer driver, a routine that specifies the routine to run when the timer hits a rollover, a routine that returns the period of the timer, and a routine that returns the frequency of the timer.</p>
INCLUDE FILES	
SEE ALSO	wvLib , <i>Wind River System Viewer User's Guide</i>

wvTsfsUploadPathLib

NAME	wvTsfsUploadPathLib – target host connection library using TSFS
ROUTINES	wvTsfsUploadPathLibInit() – initialize wvTsfsUploadPathLib library wvTsfsUploadPathCreate() – open an upload path to the host using a TSFS socket wvTsfsUploadPathClose() – close the TSFS-socket upload path wvTsfsUploadPathWrite() – write to the TSFS upload path
DESCRIPTION	<p>This library contains routines that are used by wvLib to transfer event data from the target to the host. This transfer mechanism uses the socket functionality of the Target Server File System (TSFS), and can therefore be used without including any socket or network facilities within the target.</p>
INCLUDE FILES	
SEE ALSO	wvSockUploadPathLib , wvFileUploadPathLib

xbdBlkDev

NAME	xbdBlkDev – XBD / BLK_DEV Converter
ROUTINES	xbdBlkDevLibInit() – initialize the XBD block device wrapper

xbdBlkDevCreate() – create an XBD block device wrapper
xbdBlkDevDelete() – deletes an XBD block device wrapper
xbdBlkDevCreateSync() – synchronously create an XBD block device wrapper

DESCRIPTION This library contains routines for wrapping an XBD around a **BLK_DEV**.
INCLUDE FILES **xbdBlkDev.h**

xbdCbioDev

NAME **xbdCbioDev** – XBD / CBIO Converter
ROUTINES **xbdCbioLibInit()** – initialize the XBD block device wrapper
xbdCbioDevCreate() – create an XBD CBIO device wrapper
xbdCbioDevDelete() – deletes an XBD CBIO device wrapper
DESCRIPTION This library contains routines for wrapping an XBD around a CBIO device. The CBIO device is assumed to be ready at the time the wrapper is created. No removability support is provided by this wrapper.
INCLUDE FILES **xbdCbio.h**

xbdRamDisk

NAME **xbdRamDisk** – XBD Ramdisk Implementation
ROUTINES **xbdRamDiskDevCreate()** – create an XBD ram disk
xbdRamDiskDevDelete() – XBD Ram Disk Deletion routine
DESCRIPTION This library implements an XBD ram disk.
INCLUDE FILES **xbdRamDisk.h**

xbdTrans

NAME	xbdTrans – Transaction extended-block-device
ROUTINES	formatTrans() – Format a transaction disk. transCommit() – externally-callable function to do a commit transDevCreate() – create a transactional XBD. xbdTransDevCreate() – create a transactional XBD. xbdTransInit() – initialize the transactional XBD subsystem.
DESCRIPTION	<p>This library provides device (or partition) level transaction support for XBDs. A medium with transaction support will commit changes to the media in an atomic operation (with automatical rollback if modification is not committed). As the commit operation is atomic, the media will be resistant to power failures.</p> <p>This allows synthesizing a reliable media, e.g., together with DosFS. The media will be resistant to power failures. Additionally it will always provide data integrity (different files of data on the media will always be in sync, as the application is able to set the commit points whenever the data is in sync). Note that there is a tradeoff involved: synthetic reliability is not free (in terms of performance and disk space).</p> <p>1a) How to Create a Transaction Disk on a Hard Disk / Solid State (Block/ATA) Disk (This has been designed and tested for VxWorks 6.2.)</p> <p>In most cases one should use the file system monitor, which will identify an existing transactional XBD layer and instantiate it automatically. To format a new transactional layer, use the usrFormatTrans() routine, which also works with the convenient user interfaces.</p> <p>To do this explicitly, however, first create a disk XBD, and stacking the appropriate XBD layers on top of it. For example a configuration could look like this:</p> <p>ATA XBD Block Driver XBD Transaction Layer DosFS File System</p>

A sample configuration could e.g. be created with the following code (for ATA disk 0, master, on bus 0):

```
STATUS dskinit()  
{  
    fsmNameInstall ("/ata0:1", "/ata0");  
  
    /* create xbd block device for the entire disk */  
    if ((ataXbdDevCreate (0, 0, 0, 0, "/ata0")) == (device_t) NULL)  
    {
```



```

        printErr ("ataXbdDevCreate failed.\n");
        return (ERROR);
    }

    /* Put TRFS on the device */
    if (usrFormatTrans ("/ata0",50,0) != OK)
    {
        printErr ("usrFormatTrans failed.\n");
        return (ERROR);
    }

    /* Now put dosFs on TRFS */
    if (dosFsVolFormat ("/ata0", DOS_OPT_BLANK, 0) != OK)
    {
        printErr ("Could not format for dos\n");
        return (ERROR);
    }

    /* Set a transaction point */
    usrTransCommit ("/ata0");

    return (OK);
}

```

1b) How to Create a Transaction Disk on a Floppy Driver

Do as described previously for a Hard Disk, but use the name of the floppy drive (typically "/fd0" or "A:").

1c) How to Create a Transaction Disk on a Flash (TFFS) Disk

(This has been designed and tested for VxWorks 6.2)

First create the TFFS layer to obtain a block driver. For example a configuration could look like this:

```

Flash MTD
TFFS Driver
XBD Block Driver Wrapper
XBD Transaction Layer
DosFS File System

```

As before, most operations should be done with the file system monitor. However, you must use the **tffsDevOptionsSet()** or **tffsDrvOptionsSet()** function, as shown below.

An explicit configuration could be created with the following code (for ATA disk 0, master, on bus 0):

```
device_t trans;
```

```
STATUS dskinit()
```

```
{
    /* create block device for the entire disk, */
    if (usrTffsConfig (0, 0, "/tffs") != OK)
    {
        printErr ("usrTffsConfig failed.\n");
        return (ERROR);
    }

    /* Put TRFS on the device */
    if (usrFormatTrans ("/tffs",50,0) != OK)
    {
        printErr ("usrFormatTrans failed.\n");
        return (ERROR);
    }

    /* Now put dosFs on TRFS */
    if (dosFsVolFormat ("/tffs", DOS_OPT_BLANK, 0) != OK)
    {
        printErr ("Could not format for dos\n");
        return (ERROR);
    }

    /* Set a transaction point */
    usrTransCommit ("/tffs");

    return (OK);
}
```

2) How to Commit changes to a transaction disk

The created disks are completely transaction based. This means all changes on the device have to be "committed" to actually affect any changes (note: the commit operation is designed to be atomic: it is either completely performed or not at all; if it is interrupted it is rolled back to the last successful commit). Once the transaction disk has been mounted it is immediately possible to modify its content. All modifications will be visible immediately. It is however important to understand that they have not been committed to the disk. (When power fails or system reboots, all changes are gone.)

Commits have to be manually specified. This is done by doing an `ioctl` on the highest layer (e.g., the DosFS File System). For example, this could be done by:

```

/+ flush a transaction layer to disk +/
void dskflush(char * diskname)
{
    int fd;
    fd = open(diskname,O_RDWR,0);
    ioctl(fd, FIOCOMMITFS, 0);
    close(fd);
}

```

(This code is provided in the system as **usrTransCommit()**.)

It is important to understand that:

- a) Only commit points on the disk are persistent! All changes on the disk only affect the media if they have been committed. If changes have not been committed it will automatically roll back to the last commit (reset, power fail, crash).
- b) Commit is designed to be atomic! A commit will either execute completely, or it will be rolled back to the last commit on failure (reset, power fail, crash).
- c) Commit is mainly a logical operation! Although all data will immediately be written onto the disk existing data will never be overwritten (unless a transaction is committed).
- d) Commit is essential! If the software does not commit changes the disk will never change!
- e) It is essential to commit starting from the highest layer (read on).

A commit operation has to be started from the highest layer (e.g., DosFS). This has to be done since the commit operation will traverse down the XBD stack. While traversing down it will flush any lazy write caches (to ensure everything has been written down). As a very last step it will then invoke the actual commit function in the transaction layer (which will actually write the transaction).

3) How to Format a transaction disk

A transaction disk has to be formatted independently of the file system. This can be considered kind of a "low level" format. It has to be executed in the proper sequence, e.g., for a TFFS device, you'd first "low level" format the TFFS disk (`sysTffsFormat`), then the transaction layer (`usrFormatTrans`) and then the above DosFS device (`dosFsVolFormat`).

Formatting should be done through the **usrFormatTrans()** function, which takes three parameters:

`usrTffsFormat(devname, overhead, type)`

where *devname* is the name of the disk, *overhead* is the amount of overhead space to allocate for transactions, and *type* is the media type flag. The overhead value must be between 1 (representing 0.1%) and 1000 (representing 100.0%), or 0 to use the default of 50 (representing 5%). The type flag should be one of **FORMAT_REGULAR** or **FORMAT_TFFS**.

The overhead value shrinks the logical disk drive. For instance, using the default of 50, the logical disk presented to the file system will be 5% smaller than the physical disk. Doing so ensures the ability to change files even in case of a full disk. An overhead of 50% (500) would ensure the entire disk could be modified before the transaction has to be committed (it would be possible to format a full disk, completely fill it with new data, and then either commit or rollback the complete modification).

The type is the type of format. **FORMAT_REGULAR** includes a TMR at the beginning and end of the media to increase reliability (the TMR at the beginning of the media is typically exposed to formatting problems, while the TMR at the end of the media is hard to locate as sensing the media size might fail). It should be used for all regular devices, but not **TFFS**. **FORMAT_TFFS** leaves the first sector empty (for **TFFS**) but otherwise behaves the same as **FORMAT_REGULAR**. A TMR is placed at the end of the media with a backup placed at sector 1.

4) How to get info on a transaction disk

transShow() can be used for this:

-> transShow(transXbd) Physical Information:

```
readychange= 0(0)
nBlocks=      7113(7463)
bytesPerBlk=  512(512)
```

Transaction Structure:

```
physStartOffset= 1
physTransXOffset= 1/59
physDataOffset= 117
physTransSize= 58
physDataSize= 7345
physDataUnits= 7345
physFreeUnits= 3569
physReplacedUnits= 5
physLastFreeUnit= 1207
transSerial= 9753
transNumber= 2
Disk Serial= 0x41c6/0x92c
```

Structural information: Virtual Data/Free=3771/3342 Physical

Data/Free/Replaced/Replacement=3547/3569/5/224

In detail, the provided information is:

```
readychange= 0(0)
```

readyChange status of transaction layer and underlying device (0 means layer is ready and has not detected any change, underlying layer is in bracket).

```
nBlocks=      7113(7463)
```

Total number of Blocks (Sectors) of device (transaction/logical device is smaller, see overhead, underlying device is in bracket).

bytesPerBlk= 512(512)

Bytes per Block (Sectors) of device (transaction/logical may be equal or larger than underlying device, see blkshift, underlying device is in bracket).

physStartOffset= 1

Physical Block (underlying device) start offset of disk (here in case of a TFFS disk it starts with 1 instead of 0, first block is free).

physTransXOffset= 1/59

Physical Block (underlying device) start offset of both transaction tables.

physDataOffset= 117

Physical Block (underlying device) start offset of data on the media.

physTransSize= 58

Physical Block (underlying device) size of (1) transaction table.

physDataSize= 7345

Physical Block (underlying device) size of data area.

physDataUnits= 7345

Total data units (blkshift might change this, $\text{physDataSize} = \text{physDataUnits} * 2^{\text{blkshift}}$).

physFreeUnits= 3569

Total free data units (neither containing committed nor uncommitted data).

physReplacedUnits= 5

Total replaced units. New data which would actually be written onto the media replacing data will be written into replacement units instead (it must not overwrite data already on the media). When committing this will turn into data units.

physLastFreeUnit= 1207

Last free unit. This will be used to locate the next free unit (it will typically increase monotonic).

transSerial= 9753

Monotonically increasing transaction serial number (# of commits done on the media since formatted).

transNumber= 2

Currently active transaction.

Disk Serial= 0x41c6/0x92c

Disk serial number, created upon formatting the media.

Virtual Data/Free=3771/3342

Virtual units. Data is the total number of virtual data units (including free). Free is the number of free units (from a virtual perspective). This is as the above layer (e.g. DosFS) reports it on the media.

Physical Data/Free/Replaced/Replacement=3547/3569/5/224

Physical units. Data... total number of physical data units. Free... number of free data units.

Replaced... number of replaced data units (this are units which have been changed and would have been overwritten if not replaced). Replacement... number of replacement data units (this are units used to either store new data, or replace data units when overwritten).

transShow trans,1 will display a detailed mapping plan for the device.

6) How to create rollback / completely changable disks

Setting overhead to 50% (this will only provide you with 50% of the physical disk size) will allow you to completely roll back any changes on the media w/o any limitations. This will probably hardly ever be used, probably only for small disks storing only configuration data.

7) How to activate/deactivate debug printouts

The global int transDebug = **DBG_ERR** | **DBG_STAT** /+ | **DBG_MSG** | **DBG_INFO** +/ ; can be tuned with #define **DBG_ERR** 1 #define **DBG_STAT** 2 #define **DBG_MSG** 4 #define **DBG_INFO** 8 (at runtime) to modify debug output.

8) #define PEDANTIC

This is a number of really pedantic logic tests on the media. It should be active during development and may be deactivated prior to system testing, deploying the device (although it will do no harm if installed).

Transaction logic will be heavily checked, which will significantly slow down committing on the device (especially for large disks or slow CPUs).

When a logic error is detected it will typically trigger **transPanic()**, resulting in calling the user hook and deactivating the complete transaction layer (for safety reasons, as the disk should no longer be modified).

9) **transPanic()**

Panics are typically fatal. Although a hook is provided to "catch" panics the hook will NOT allow to recover from them.

Types are:

ERROR_LOCK error acquiring the lock (**semTake()** error This is a fatal internal error. The semaphore operation has returned an error.

ERROR_TRANS_WRITE error writing transaction Transaction table write operations are generally considered fatal. The layer/ disk/ device has a major problem. If the write has failed it is likely that data from non failing writes will not be written correct. The layer goes therefore down.

ERROR_CONSISTENCY internal consistency check error Internal consistency checking found a major problem. This is most likely a media independend problem, the layer itself has encountered a bug. It has to go down.

ERROR_IO_OPERATION general IO error (underlying layer should not return error!) General read or write error when accessing the underlying layer (during a data read/write operation). This should never happen. Please be aware that the layer will only try to read

data from previously completed operations (it can never read data blocks from previously interrupted operations). This problem can therefore not be caused by e.g. a NAND flash which has been interrupted while writing (ECC error).

ERROR_OUTOFMEMORY out of heap memory Malloc returned NULL.

ERROR_UNCATCHED system was unable to recover from warning A warning (see "What is **transWarning()**?") that should have been caught has not been caught and is therefore considered fatal.

ERROR_OUT_OF_UNITS out of physical units The system has run out of free space. This should not happen as a warning **WARN_OUT_OF_UNITS** has been triggered previously... and should either lead to a fatal error **ERROR_UNCATCHED**, or should have been resolved. As this should not happen it is fatal.

ERROR_DEFRAG error while defragmenting Problem while in the still pretty experimental deframer. It is fatal as further modifications might lead to a corrupt disk.

On a panic the XBD device will disable itself. It will do so by ejecting itself; at this point the XBD is no longer useable. This is done to ensure the transaction layer can no longer be used (several panics are from internal errors, it makes sense to make absolutely sure the layer goes down).

10) **transWarning()**

Most triggered warnings should be considered fatal. A hook is provided to catch those warnings; additionally the hook allows recovering from warnings (just continue processing and ignore the warning). A triggered warning should however be considered with certain severity.

Types are:

WARN_OUT_OF_UNITS See below for details. This should be caught and considered serious if not specifically intended otherwise.

WARN_TRANS_CRC_MIS This should typically never happen. The transaction head and tail have been completely verified (and are correct). Nevertheless the transaction data does not match the CRC. This can only mean the transaction data has been corrupted AFTER writing. The system would now try to find another transaction (would revert to the second transaction). The old transaction might however no longer be valid (after the commit the system is allowed to replace data for the first time). Therefore, the other transaction should NOT be used for this special condition! This is automatically caught and converted into a panic (uncaught warning). It can be caught by the user and converted into OK, but this involves above mentioned risk and might not be advisable!

All uncaught warnings (but **WARN_OUT_OF_UNITS**) are automatically converted to panics (see "What is **transPanic()**?")

11) **WARN_OUT_OF_UNITS**

There are certain limitations on the size of new or replaced information you can store on a volume. Consider a volume of 100MB physical size. It is formatted with a standard of 5% (50)

overhead, which provides you with a 95MB logical volume. If this disk is filled up with 80MB of data you have only 15MB of free space. Of course you won't have a limit when adding new files (other than the 15MB limit which is obvious). When you however start replacing files you can reach a new limit: the limit of the transaction layer. When you replace/overwrite a file of 15MB size everything is fine. You can replace the file and commit after this has been done. When you however replace a file of 21MB (or more) size you'd essentially try to write 101MB of data onto the volume. This will of course fail.

The system will then call the warning **transWarning() WARN_OUT_OF_UNITS**. This will print a warning on the console, but otherwise continue. If the warning hook does nothing, the system will fail the I/O and eject the XBD (effectively rolling back the uncommitted transaction).

12) Why the system is resistant to (NAND) ECC/CRC errors

There are 3 distinct types of data on the disk:

TMR (Transaction Master Record) This is only written once when formatting the media. It is therefore not in risk at a power fail.

Transaction Table Two copies of this exist. The layer is hardened to ignore read errors (and revert to the other copy if it makes sense).

Data Only data that has been completely written to the media will ever be read. While a power fail/error might occur during a data write, this data block will not be committed and can therefore never be read. Only after a complete commit the data has been made accessible.

13) How long will it take to commit

Commit time will depend on Blocks Cached in upper and lower layers, plus number of changes. A minimum commit consists of: no data dirty in caches, therefore nothing to flush 3 sectors written (transaction head + tail + 1 transaction data sector changed).

Transaction data sectors are only written if they have changed, the code maintains the dirty status on both tables.

14) Disk and RAM memory consumption

a) **Disk Space** The transaction layer will introduce the following overhead in terms of disk space on the media:

- + **Transaction Master Record (TMR)** TMR has a similar function than the Master Boot Record (MBR) on a FAT device. It is used to identify the media, and to obtain its basic parameters. Depending on the chosen format (there are 3 different types of format to be chosen when the media is initially formatted) the MBR will take 1 or 2 physical disk sectors.

- + **Transaction** A transaction keeps track of the logical to physical unit mapping. A media has exactly 2 transaction records. A transaction consists of a transaction head followed by transaction data followed by a transaction tail. Transaction head and tail are used to identify the version and the integrity of the transaction, they will always take exactly 1 physical disk sector each. The size of the transaction data depends on the number of logical units the

device provides. A logical unit is the internal view for grouping 2^n sectors together (The transaction layer can increase the block size presented to the upper layer. This can be done to decrease memory overhead both on the disk and in RAM, or to further increase performance. This is controlled by the blkshift, which is chosen when formatting the media: $\text{sectSizeUpper} = \text{unitsize} = \text{sectSizeLower} * 2^{\text{blkshift}}$). Both transactions including head and tail will take 4 physical disk sectors +

Memory is approx: Memory(RAM&Disk) is linear with $1/(2^{\text{blkshift}})$; (Doubling the sector size will half the memory overhead).

+ Configured overhead Overhead is a chosen value. It can be used to increase the rollback capability of the media (the amount of data that can possibly be "rolled back" is of course determined by a logical boundary: the amount of free media on the device. A completely filled up media would therefore become read-only if there is no extra space provided). The recommended value for this is 50promille = 5%. The overhead actually created is percentage of overhead - Transaction Size - TMR size (overhead is calculated in a way that it expresses the complete amount of space volunteered to the transaction layer... and overhead of e.g. 0 is therefore not possible).

Memory (RAM) consumption: The current version requires one complete transaction to be stored in RAM (additionally a few bitfields are used which cause a minor additional overhead). The exact memory consumption is:

(based on number of units... a unit is a logical (upper) block. Number and size of units can be controlled by using blkshift)

32bit per (#units) for logical2physical mapping table (transaction data) 2bit per (#units * 4 byte / physical block size) for dirty bits
(for both transactions)

2bit per (physical #units) for block status information

This roughly translates into:

$4.25\text{bytes} * (1/(2^{\text{blkshift}})) * (\# \text{ blocks on media})$

INCLUDE FILES **transCbio.h**

SEE ALSO *VxWorks Programmers Guide: I/O System*

