WIND RIVER

VxWorks®

KERNEL API REFERENCE

Volume 2: Routines

6.6

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

VxWorks Kernel API Reference, Volume 2: Routines, 6.6

15 Nov 07
Part #: DOC-16102-ND-00

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 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 (this book) provides reference entries for each of the routines found in the VxWorks kernel libraries documented in Volume 1.

Volume 2

Routines

CPToUtf16( ) – Convert a Unicode codepoint to a UTF-16 encoding **41**
CPToUtf8( ) – Convert a Unicode codepoint to a UTF-8 encoding **41**
CPUSET_ATOMICCLR( ) – atomically clear a CPU from a CPU set **42**
CPUSET_ATOMICCOPY( ) – atomically copy a CPU set value **42**
CPUSET_ATOMICSET( ) – atomically set a CPU in a CPU set **43**
CPUSET_CLR( ) – clear a CPU from a CPU set **44**
CPUSET_ISSET( ) – determine if a CPU is set in a CPU set **44**
CPUSET_ISZERO( ) – determine if all CPUs are cleared from a CPU set **45**
CPUSET_SET( ) – set a CPU in a CPU set **45**
CPUSET_SETALL( ) – set all CPUs in a CPU set **46**
CPUSET_SETALL_BUT_SELF( ) – set all CPUs except self in CPU set **46**
CPUSET_ZERO( ) – clear all CPUs from a CPU set **47**
Sysctl( ) – get or set values for kernel state variables from the C shell **48**
VX_MEM_BARRIER_R( ) – Read Memory Barrier **49**
VX_MEM_BARRIER_RW( ) – Read/Write Memory Barrier **50**
VX_MEM_BARRIER_W( ) – Write memory barrier **51**
a0( ) – return the contents of register **a0** (also **a1** - **a7**) (MC680x0) **51**
access( ) – determine accessibility of a file **52**
acosf( ) – compute an arc cosine (ANSI) **53**
adrSpaceInfoGet( ) – get status of the address space library **53**
adrSpacePageUnmap( ) – unmap a set of virtual pages **54**
adrSpaceRAMAddToPool( ) – add specified memory block to RAM pool **55**
adrSpaceRAMReserve( ) – reserve memory from the RAM pool **56**
adrSpaceShow( ) – display information about address spaces managed by **adrSpaceLib** **57**
adrSpaceVirtReserve( ) – reserve memory from the virtual space **58**
aimCacheInit( ) – initialize cache aim with supplied parameters **59**
aimFppLibInit( ) – Initialize the AIM FPU library **59**
aimMmuLibInit( ) – initialize the AIM **60**
aioShow( ) – show AIO requests **60**
aioSysInit( ) – initialize the AIO system driver **61**

aio_cancel( ) – cancel an asynchronous I/O request (POSIX) **61**
aio_error( ) – retrieve error status of asynchronous I/O operation (POSIX) **62**
aio_fsync( ) – asynchronous file synchronization (POSIX) **63**
aio_read( ) – initiate an asynchronous read (POSIX) **63**
aio_return( ) – retrieve return status of asynchronous I/O operation (POSIX) **64**
aio_suspend( ) – wait for asynchronous I/O request(s) (POSIX) **65**
aio_write( ) – initiate an asynchronous write (POSIX) **65**
alarm( ) – set an alarm clock for delivery of a signal **66**
anRegister( ) – register with the VxBus subsystem **67**
asinf( ) – compute an arc sine (ANSI) **67**
atan2f( ) – compute the arc tangent of y/x (ANSI) **67**
atanf( ) – compute an arc tangent (ANSI) **68**
atapiParamsPrint( ) – Print the drive parameters. **68**
attrib( ) – modify MS-DOS file attributes on a file or directory **69**
b( ) – set or display breakpoints **70**
bcmp( ) – compare one buffer to another **71**
bcopy( ) – copy one buffer to another **71**
bcopyBytes( ) – copy one buffer to another one byte at a time **72**
bcopyLongs( ) – copy one buffer to another one long word at a time **72**
bcopyWords( ) – copy one buffer to another one word at a time **73**
bd( ) – delete a breakpoint **73**
bdall( ) – delete all breakpoints **74**
bfStrSearch( ) – Search using the Brute Force algorithm **75**
bfill( ) – fill a buffer with a specified character **75**
bfillBytes( ) – fill buffer with a specified character one byte at a time **76**
bh( ) – set a hardware breakpoint **76**
binvert( ) – invert the order of bytes in a buffer **77**
bmsStrSearch( ) – Search using the Boyer-Moore-Sunday (Quick Search) algorithm **77**
bmtPhyRegister( ) – register with the VxBus subsystem **78**
bootBpAnchorExtract( ) – extract a backplane address from a device field **79**
bootChange( ) – change the boot line **79**
bootLeaseExtract( ) – extract the lease information from an Internet address **80**
bootNetmaskExtract( ) – extract the net mask field from an Internet address **81**
bootParamsPrompt( ) – prompt for boot line parameters **82**
bootParamsShow( ) – display boot line parameters **82**
bootStringToStruct( ) – interpret the boot parameters from the boot line **83**
bootStringToStructAdd( ) – interpret the boot parameters from the boot line **83**
bootStructToString( ) – construct a boot line **84**
bswap( ) – swap buffers **84**
bzero( ) – zero out a buffer **85**
c( ) – continue from a breakpoint **85**
cacheArchClearEntry( ) – clear an entry from a cache (68K, x86) **86**
cacheArchLibInit( ) – initialize the cache library **87**
cacheAuLibInit( ) – initialize the Au cache library **88**
cacheClear( ) – clear all or some entries from a cache **89**

cacheDisable( ) – disable the specified cache **89**
cacheDmaFree( ) – free the buffer acquired with **cacheDmaMalloc( )** **90**
cacheDmaMalloc( ) – allocate a cache-safe buffer for DMA devices and drivers **90**
cacheDrvFlush( ) – flush the data cache for drivers **91**
cacheDrvInvalidate( ) – invalidate data cache for drivers **91**
cacheDrvPhysToVirt( ) – translate a physical address for drivers **92**
cacheDrvVirtToPhys( ) – translate a virtual address for drivers **92**
cacheEnable( ) – enable the specified cache **93**
cacheFlush( ) – flush all or some of a specified cache **93**
cacheForeignClear( ) – clear foreign data from selected cache **94**
cacheForeignFlush( ) – flush foreign data from selected cache **94**
cacheForeignInvalidate( ) – invalidate foreign data from selected cache **95**
cacheInvalidate( ) – invalidate all or some of a specified cache **95**
cacheLibInit( ) – initialize the cache library for a processor architecture **96**
cacheLock( ) – lock all or part of a specified cache **96**
cachePipeFlush( ) – flush processor write buffers to memory **97**
cacheR10kLibInit( ) – initialize the R10000 cache library **97**
cacheR4kLibInit( ) – initialize the R4000 cache library **98**
cacheR5kLibInit( ) – initialize the R5000 cache library **99**
cacheR7kLibInit( ) – initialize the R7000 cache library **99**
cacheSh7750LibInit( ) – initialize the SH7750 cache library **100**
cacheStoreBufDisable( ) – disable the store buffer (MC68060 only) **102**
cacheStoreBufEnable( ) – enable the store buffer (MC68060 only) **102**
cacheTextLocalUpdate( ) – synchronize the caches on local cpu only **102**
cacheTextUpdate( ) – synchronize the instruction and data caches **103**
cacheTx49LibInit( ) – initialize the Tx49 cache library **103**
cacheUnlock( ) – unlock all or part of a specified cache **104**
calloc( ) – allocate space for an array (ANSI) **105**
cbioBlkCopy( ) – block to block (sector to sector) transfer routine **105**
cbioBlkRW( ) – transfer blocks to or from memory **106**
cbioBytesRW( ) – transfer bytes to or from memory **106**
cbioDevCreate( ) – Initialize a CBIO device (Generic) **107**
cbioDevVerify( ) – verify **CBIO_DEV_ID** **108**
cbioIoctl( ) – perform ioctl operation on device **108**
cbioLibInit( ) – Initialize CBIO Library **109**
cbioLock( ) – obtain CBIO device semaphore. **110**
cbioModeGet( ) – return the mode setting for CBIO device **110**
cbioModeSet( ) – set mode for CBIO device **111**
cbioParamsGet( ) – fill in **CBIO_PARAMS** structure with CBIO device parameters **111**
cbioRdyChgdGet( ) – determine ready status of CBIO device **112**
cbioRdyChgdSet( ) – force a change in ready status of CBIO device **113**
cbioShow( ) – print information about a CBIO device **113**
cbioUnlock( ) – release CBIO device semaphore. **114**
cbioWrapBlkDev( ) – create CBIO wrapper atop a **BLK_DEV** device **114**
cbrt( ) – compute a cube root **115**

cbrtf( ) – compute a cube root 115
cd( ) – change the default directory 116
cdromFsDevCreate( ) – create a CD-ROM filesystem (cdromFs) I/O device. 117
cdromFsDevDelete( ) – delete a CD-ROM filesystem (cdromFs) I/O device 118
cdromFsInit( ) – initialize the VxWorks CD-ROM file system 118
cdromFsVersionDisplay( ) – display the cdromFs version number 119
cdromFsVersionNumGet( ) – return the cdromFs version number 119
cdromFsVolConfigShow( ) – show the volume configuration information 119
ceilf( ) – compute the smallest integer greater than or equal to a specified value (ANSI) 120
cfree( ) – free a block of memory from the system memory partition (kernel heap) 120
checkStack( ) – print a summary of each task's stack usage 121
chkdsk( ) – perform consistency checking on a MS-DOS file system 122
chmod( ) – change the permission mode of a file 123
clock_getres( ) – get the clock resolution (POSIX) 125
clock_gettime( ) – get the current time of the clock (POSIX) 125
clock_nanosleep( ) – high resolution sleep with specifiable clock 126
clock_setres( ) – set the clock resolution 127
clock_settime( ) – set the clock to a specified time (POSIX) 127
close( ) – close a file 128
closedir( ) – close a directory (POSIX) 128
cnsAppRegister( ) – Registers an application with the CNS library. 129
cnsClose( ) – Close or create and open named communication medium for read/write. 130
cnsCompLibInit( ) – Initialize the CNS COMP library. 130
cnsDefaultMediaTypeSet( ) – Set the default media type. 131
cnsLibInit( ) – Initialize the CNS library. 132
cnsMediaRegister( ) – Registers a communication media with the CNS. 132
cnsMediaTypeRemove( ) – Remove a media type from an application's media list. 133
cnsMediumTypeNext( ) – Return the name of the media type next in the list. 134
cnsMsgEncode( ) – Encode a message as understood by CNS. 134
cnsOpen( ) – Open or create and open named communication medium for read/write. 136
cnsRead( ) – Read from a communication medium. 136
cnsWrite( ) – Write to a communication medium. 137
commit( ) – commit current transaction to disk. 138
copy( ) – copy *in* (or stdin) to *out* (or stdout) 139
copyStreams( ) – copy from/to specified streams 140
coreDumpClose( ) – close a core dump 140
coreDumpCopy( ) – copy a core dump to the given path 141
coreDumpCreateHookAdd( ) – add a routine to be called at every core dump create 141
coreDumpCreateHookDelete( ) – delete a previously added core dump create routine 142
coreDumpDevFormat( ) – format the core dump device 142
coreDumpDevShow( ) – display information on core dump device 143
coreDumpInfoGet( ) – get information on a core dump 143
coreDumpIsAvailable( ) – is a core dump available for retrieval 145
coreDumpMemDump( ) – dump an area of memory in VxWorks core dump 145
coreDumpMemFilterAdd( ) – add a memory region filter 146

coreDumpMemFilterDelete( ) – delete a memory region filter **147**
coreDumpNextGet( ) – get the next core dump on device **147**
coreDumpOpen( ) – open an existing core dump for retrieval **148**
coreDumpRead( ) – read from a core file **148**
coreDumpShow( ) – display information on core dumps **149**
coreDumpUsrGenerate( ) – generate a user (on-demand) core dump **150**
cosf( ) – compute a cosine (ANSI) **151**
coshf( ) – compute a hyperbolic cosine (ANSI) **151**
cp( ) – copy file into other file/directory. **152**
cplusCallNewHandler( ) – call the allocation failure handler (C++) **152**
cplusCtors( ) – call static constructors (C++) **153**
cplusCtorsLink( ) – call all linked static constructors (C++) **154**
cplusDemanglerSet( ) – change C++ demangling mode (C++) **154**
cplusDemanglerStyleSet( ) – change C++ demangling style (C++) **155**
cplusDtors( ) – call static destructors (C++) **155**
cplusDtorsLink( ) – call all linked static destructors (C++) **156**
cplusLibInit( ) – initialize the C++ library (C++) **157**
cplusXtorGet( ) – get the c++ Xtors strategy **157**
cplusXtorSet( ) – change C++ static constructor calling strategy (C++) **158**
cpsr( ) – return the contents of the current processor status register (ARM) **158**
cpuPwrMgrEnable( ) – Set the CPU light power management to ON/OFF **159**
cpuPwrMgrIsEnabled( ) – Get the CPU power management status **160**
creat( ) – create a file **160**
cret( ) – continue until the current subroutine returns **161**
d( ) – display memory **162**
d0( ) – return the contents of register **d0** (also **d1** - **d7**) (MC680x0) **162**
dbgBpTypeBind( ) – bind a breakpoint handler to a breakpoint type (MIPS R3000, R4000, R4650) **163**
dbgHelp( ) – display debugging help menu **163**
dbgInit( ) – initialize the shell debugging package **164**
dcacheDevCreate( ) – Create a disk cache **165**
dcacheDevDisable( ) – Disable the disk cache for this device **165**
dcacheDevEnable( ) – Reenable the disk cache **166**
dcacheDevMemResize( ) – set a new size to a disk cache device **166**
dcacheDevTune( ) – modify tunable disk cache parameters **167**
dcacheHashTest( ) – test hash table integrity **168**
dcacheShow( ) – print information about disk cache **169**
devs( ) – list all system-known devices **169**
dirList( ) – list contents of a directory (multi-purpose) **170**
diskFormat( ) – format a disk with dosFs **170**
diskInit( ) – initialize a file system on a block device **171**
dosFsCacheCreate( ) – create cache for a DosFS volume **171**
dosFsCacheDelete( ) – delete the disk cache for a dosFs volume **172**
dosFsCacheInfo( ) – retrieve a cache's settings **172**
dosFsCacheLibInit( ) – initialize dosFsCache library. **173**
dosFsCacheOptionsGet( ) – get this dosFs volume's disk cache options **174**

dosFsCacheOptionsSet( ) – set this dosFs volume's disk cache options **174**
dosFsCacheShow( ) – show information regarding a dosFs volume's cache **175**
dosFsCacheTune( ) – tune a cache's settings **175**
dosFsChkDsk( ) – make volume integrity checking. **176**
dosFsClose( ) – close a dosFs file **176**
dosFsDefaultCacheSizeSet( ) – set the default disk cache size **177**
dosFsDefaultDataCacheSizeGet( ) – get the default data cache size **177**
dosFsDefaultDirCacheSizeGet( ) – get the default directory cache size **178**
dosFsDefaultFatCacheSizeGet( ) – get the default FAT cache size **178**
dosFsDevCreate( ) – create file system device. **178**
dosFsDevDelete( ) – delete a dosFs volume **180**
dosFsDiskProbe( ) – probe if a device contains a valid dosFs **180**
dosFsFdFree( ) – free a file descriptor **181**
dosFsFdGet( ) – get an available file descriptor **181**
dosFsFmtLibInit( ) – initialize the MS-DOS formatting library **182**
dosFsFmtTest( ) – UNITEST CODE **182**
dosFsHdlrInstall( ) – install handler. **183**
dosFsIoctl( ) – do device specific control function **183**
dosFsLastAccessDateEnable( ) – enable last access date updating for this volume **184**
dosFsLibInit( ) – prepare to use the dosFs library **185**
dosFsMonitorDevCreate( ) – create a dosFs volume through the fs monitor **185**
dosFsOpen( ) – open a file on a dosFs volume **186**
dosFsShow( ) – display dosFs volume configuration data. **186**
dosFsVolDescGet( ) – convert a device name into a DOS volume descriptor pointer. **187**
dosFsVolFormat( ) – format an MS-DOS compatible volume **188**
dosFsVolFormatFd( ) – format an MS-DOS compatible volume via an opened FD **189**
dosFsVolIsFat12( ) – determine if a MSDOS volume is FAT12 or FAT16 **190**
dosFsVolUnmount( ) – unmount a dosFs volume **192**
dosFsVolumeOptionsGet( ) – get this volume's disk options **193**
dosFsVolumeOptionsSet( ) – set this volume's disk options **193**
dosFsXbdBlkCopy( ) – copy blocks on the underlying XBD block device. **194**
dosFsXbdBlkRead( ) – read blocks from the underlying XBD block device. **195**
dosFsXbdBlkWrite( ) – write blocks to the underlying XBD block device. **195**
dosFsXbdBytesRW( ) – read/write bytes to/from the underlying XBD block device. **196**
dosFsXbdIoctl( ) – Misc control operations **196**
dosPathParse( ) – parse a full pathname into an array of names. **197**
dosSetVolCaseSens( ) – set case sensitivity of volume **198**
dosfsDiskFormat( ) – format a disk with dosFs **198**
dosfsDiskToHost16( ) – convert uint16_t from on-disk to host format **199**
dosfsDiskToHost32( ) – convert uint32_t from on-disk to host format **199**
dosfsHostToDisk16( ) – convert uint16_t from host to on-disk format **200**
dosfsHostToDisk32( ) – convert uint32_t from host to on-disk format **200**
dpartDevCreate( ) – Initialize a partitioned disk **201**
dpartPartGet( ) – retrieve handle for a partition **202**
dshmMuxHwAddrToOff( ) – translate a local address to a shared memory offset **202**

dshmMuxHwGet( ) – obtain an hardware registration handle based on name **203**
dshmMuxHwLocalAddrGet( ) – obtain address of the local node **203**
dshmMuxHwNodesNumGet( ) – obtain the maximum number of nodes on a hardware bus **204**
dshmMuxHwOffToAddr( ) – translate a shared memory offset to a local address **204**
dshmMuxHwRegister( ) – register a hardware bus with the MUX **205**
dshmMuxHwTasClearGet( ) – obtain the TAS clear routine on this bus **205**
dshmMuxHwTasGet( ) – obtain the test-and-set routine on this bus **206**
dshmMuxLibInit( ) – initialize the DSHM MUX **206**
dshmMuxMemAlloc( ) – allocate shared memory from a specific hardware **207**
dshmMuxMemFree( ) – free allocated shared memory from a specific hardware **208**
dshmMuxMsgRecv( ) – receive a message **208**
dshmMuxMsgSend( ) – transmit a message **209**
dshmMuxSvcNodeJoin( ) – signal services that a node has joined the system **209**
dshmMuxSvcNodeLeave( ) – signal services that a node has left the system **210**
dshmMuxSvcObjGet( ) – retrieve a service object and protect it against deletion **210**
dshmMuxSvcObjRelease( ) – allows modifications to be made on a service object **211**
dshmMuxSvcRegister( ) – register a service with the MUX **211**
dshmMuxSvcWithdraw( ) – remove service from MUX **212**
dshmMuxWidtdrawComplete( ) – signal service has finished withdrawing **212**
dsiDataPoolShow( ) – display DSI's data pool statistics **213**
dsiSysPoolShow( ) – display DSI's system pool statistics **213**
e( ) – set or display eventpoints (WindView) **214**
edi( ) – return the contents of register **edi** (also **esi** - **eax**) (x86) **215**
edi( ) – return the contents of register **edi** (also **esi** - **eax**) (x86/SimNT) **215**
edrBootCountGet( ) – returns the current boot count **216**
edrBootShow( ) – show all stored boot type ED&R records **216**
edrClear( ) – a synonym for edrErrorLogClear **217**
edrErrLogAttach( ) – attach to an existing log **217**
edrErrLogClear( ) – clear the log's contents **217**
edrErrLogCreate( ) – create a new log **218**
edrErrLogIterCreate( ) – create an iterator for traversing the log **219**
edrErrLogIterNext( ) – returns the next committed node **219**
edrErrLogMaxNodeCount( ) – return the maximum number of nodes in the log **220**
edrErrLogNodeAlloc( ) – allocate a node from the error log **220**
edrErrLogNodeCommit( ) – commits a previously allocated node **221**
edrErrLogNodeCount( ) – return the number of committed nodes in the log **221**
edrErrorInject( ) – injects an error into the ED&R subsystem **222**
edrErrorInjectHookAdd( ) – adds a hook which gets called on error-injection **223**
edrErrorInjectHookDelete( ) – removes an existing error-inject hook **224**
edrErrorInjectPrePostHookAdd( ) – adds a hook which gets called before and after error-injection **225**
edrErrorInjectPrePostHookDelete( ) – removes the existing pre/post hook **226**
edrErrorInjectTextHookAdd( ) – adds a hook which gets called on record creation **226**
edrErrorInjectTextHookDelete( ) – removes the existing text writing hook **227**
edrErrorLogClear( ) – clears the ED&R error log **227**
edrErrorRecordCount( ) – returns the number of error-records in the log **228**

edrErrorRecordDecode( ) – decode one error-record **228**
edrFatalShow( ) – show all stored fatal type ED&R records **229**
edrFlagsGet( ) – return the ED&R flags which are currently set **229**
edrHelp( ) – prints helpful information on ED&R **230**
edrHookShow( ) – show the list of installed ED&R hook routines **230**
edrInfoShow( ) – show all stored info type ED&R records **230**
edrInitShow( ) – show all stored init type ED&R records **231**
edrInjectHookShow( ) – show the list of error injection hook routines **231**
edrInjectPrePostHookShow( ) – show the list of pre/post injection hook routines **232**
edrInjectTextHookShow( ) – show the list of text injection hook routines **232**
edrIntShow( ) – show all stored interrupt type ED&R records **233**
edrIsDebugMode( ) – is the ED&R debug mode flag set? **233**
edrKernelShow( ) – show all stored kernel type ED&R records **233**
edrLibInit( ) – initializes **edrLib** **234**
edrRebootShow( ) – show all stored reboot type ED&R records **235**
edrRtpShow( ) – show all stored rtp type ED&R records **235**
edrShow( ) – displays the ED&R error log to stdout **236**
edrSystemDebugModeGet( ) – indicates if the system is in **debug** mode **236**
edrSystemDebugModeInit( ) – initialise the system mode debug flag **237**
edrSystemDebugModeSet( ) – modifies the system **debug** mode flag **237**
edrUserShow( ) – show all stored user type ED&R records **238**
eflags( ) – return the contents of the status register (x86) **238**
eflags( ) – return the contents of the status register (x86/SimNT) **239**
elPciRegister( ) – register with the VxBus subsystem **239**
eneRegister( ) – register with the VxBus subsystem **239**
envGet( ) – return a pointer to the environment of a task **240**
envLibInit( ) – initialize environment variable facility **240**
envPrivateCreate( ) – create a private environment **241**
envPrivateDestroy( ) – destroy a private environment **242**
envShow( ) – display the environment for a task **242**
errnoGet( ) – get the error status value of the calling task **243**
errnoOfTaskGet( ) – get the error status value of a specified task **243**
errnoOfTaskSet( ) – set the error status value of a specified task **244**
errnoSet( ) – set the error status value of the calling task **244**
etsecRegister( ) – register with the VxBus subsystem **245**
eventClear( ) – Clear the calling task's events register **245**
eventReceive( ) – Wait for event(s) **245**
eventSend( ) – Send event(s) **247**
excConnect( ) – connect a C routine to an exception vector (PowerPC) **248**
excCrtConnect( ) – connect a C routine to a critical exception vector (PowerPC 403) **249**
excHookAdd( ) – specify a routine to be called with exceptions **250**
excInit( ) – initialize the exception handling package **251**
excIntConnect( ) – connect a C routine to an asynchronous exception vector (PowerPC, ARM) **251**
excIntCrtConnect( ) – connect a C routine to a critical interrupt vector (PowerPC 403) **252**
excJobAdd( ) – request a task-level function call from interrupt level **253**

excVecGet( ) – get a CPU exception vector (PowerPC, ARM) **253**
excVecInit( ) – initialize the exception/interrupt vectors **254**
excVecSet( ) – set a CPU exception vector (PowerPC, ARM) **255**
exit( ) – exit a task (ANSI) **256**
expf( ) – compute an exponential value (ANSI) **256**
fabsf( ) – compute an absolute value (ANSI) **257**
fastStrSearch( ) – Search by optimally choosing the search algorithm **257**
fccRegister( ) – register with the VxBus subsystem **258**
fchmod( ) – change the permission mode of a file **258**
fcntl( ) – perform control functions over open files **259**
fdatasync( ) – synchronize a file data **260**
fdprintf( ) – write a formatted string to a file descriptor **261**
fecRegister( ) – register with the VxBus subsystem **261**
feiRegister( ) – register with the VxBus subsystem **262**
ffsLsb( ) – find least significant bit set **262**
ffsMsb( ) – find most significant bit set **262**
fileUploadPathClose( ) – close the event-destination file **263**
fioBaseLibInit( ) – initialize the formatted I/O support library **263**
fioFormatV( ) – convert a format string **264**
fioLibInit( ) – initialize the formatted I/O support library **265**
fioRdString( ) – read a string from a file **265**
fioRead( ) – read a buffer **266**
floorf( ) – compute the largest integer less than or equal to a specified value (ANSI) **266**
fmodf( ) – compute the remainder of x/y (ANSI) **267**
formatTrans( ) – Format a transaction disk. **267**
fpathconf( ) – determine the current value of a configurable limit **268**
fppInit( ) – initialize floating-point coprocessor support **269**
fppProbe( ) – probe for the presence of a floating-point coprocessor **269**
fppRestore( ) – restore the floating-point coprocessor context **270**
fppSave( ) – save the floating-point coprocessor context **271**
fppShowInit( ) – initialize the floating-point show facility **273**
fppTaskRegsGet( ) – Gets FPU context for a task **273**
fppTaskRegsGet( ) – get the floating-point registers from a task TCB **274**
fppTaskRegsSet( ) – Sets FPU context for a task **274**
fppTaskRegsSet( ) – set the floating-point registers of a task **275**
fppTaskRegsShow( ) – print the contents of a task's floating-point registers **275**
free( ) – free a block of memory from the system memory partition (ANSI) **276**
fsEventUtilInit( ) – Initialize the file system event utlility library **276**
fsMonitorInit( ) – Initialize the fsMonitor **277**
fsPathAddedEventRaise( ) – Raise a "path added" event **277**
fsPathAddedEventSetup( ) – Setup to wait for a path **278**
fsWaitForPath( ) – wait for a path **278**
fsmGetDriver( ) – Get the XBD name of a mapping based on the path **279**
fsmGetVolume( ) – get the pathname based on an XBD name mapping **279**
fsmNameInstall( ) – Add a mapping between an XBD name and a pathname **280**

fsmNameMap( ) – map an XBD name to a Core I/O path **280**
fsmNameUninstall( ) – remove an XBD name to pathname mapping **281**
fsmProbeInstall( ) – install F/S probe and instantiator functions **281**
fsmProbeUninstall( ) – remove a file system probe **282**
fsmUnmountHookAdd( ) – Add an unmount hook function **283**
fsmUnmountHookDelete( ) – Remove an unmount hook function **283**
fsmUnmountHookRun( ) – Runs the unmount hook functions **284**
fstat( ) – get file status information (POSIX) **284**
fstatfs( ) – get file status information (POSIX) **285**
fsync( ) – synchronize a file **286**
ftruncate( ) – truncate a file (POSIX) **286**
g0( ) – return the contents of register g0 (also g1-g7) (SimSolaris) **287**
geiRegister( ) – register with the VxBus subsystem **287**
getOptServ( ) – parse parameter string into argc, argv format **288**
getenv( ) – get an environment variable (ANSI) **288**
getopt( ) – parse argc/argv argument vector (POSIX) **289**
getoptInit( ) – initialize the getopt state structure **290**
getopt_r( ) – parse argc/argv argument vector (POSIX) **291**
h( ) – display or set the size of shell history **292**
hashFuncIterScale( ) – iterative scaling hashing function for strings **293**
hashFuncModulo( ) – hashing function using remainder technique **293**
hashFuncMultiply( ) – multiplicative hashing function **294**
hashKeyCmp( ) – compare keys as 32 bit identifiers **294**
hashKeyStrCmp( ) – compare keys based on strings they point to **295**
hashTblCreate( ) – create a hash table **295**
hashTblDelete( ) – delete a hash table **296**
hashTblDestroy( ) – destroy a hash table **297**
hashTblEach( ) – call a routine for each node in a hash table **297**
hashTblFind( ) – find a hash node that matches the specified key **298**
hashTblInit( ) – initialize a hash table **298**
hashTblPut( ) – put a hash node into the specified hash table **299**
hashTblRemove( ) – remove a hash node from a hash table **300**
hashTblTerminate( ) – terminate a hash table **300**
help( ) – print a synopsis of selected routines **301**
histLoad( ) – load history into the current shell session interpreter(s) **302**
histSave( ) – save history of the current shell session interpreter(s) **302**
hookAddToHead( ) – add a hook routine at the start of a hook table **303**
hookAddToTail( ) – add a hook routine to the end of a hook table **304**
hookDelete( ) – delete a hook from a hook table **304**
hookFind( ) – Search a hook table for a given hook **305**
hookShow( ) – show the hooks in the given hook table **305**
hrfsAdvFormat( ) – format the HRFS file system using advanced options **306**
hrfsAdvFormatFd( ) – format the HRFS file system using advanced options via a file descriptor **307**
hrfsAscTime( ) – convert "broken-down" HRFS time to string **308**
hrfsChkDsk( ) – check the HRFS file system **308**

hrfsDevCreate( ) – create an HRFS device **309**
hrfsDiskFormat( ) – format a disk with HRFS **310**
hrfsFormat( ) – format the HRFS file system via a path **310**
hrfsFormatFd( ) – format the HRFS file system via a file descriptor **311**
hrfsFormatLibInit( ) – prepare to use the HRFS formatter **312**
hrfsTimeCondense( ) – condense time in **HRFS_TM** to time in msec **313**
hrfsTimeGet( ) – return # of milliseconds since midnight Jan 1, 1970 **313**
hrfsTimeSplit( ) – split time in msec into **HRFS_TM** format **313**
hrfsUpgrade( ) – upgrade the HRFS file system to the latest version **314**
i( ) – print a summary of each task's TCB **315**
i0( ) – return the contents of register i0 (also i1-i7) (SimSolaris) **316**
i8042vxbRegister( ) – register i8042vxb driver **316**
ichAtaBlkRW( ) – read or write sectors to a ATA/IDE disk. **317**
ichAtaCmd( ) – issue a RegisterFile command to ATA/ATAPI device. **317**
ichAtaConfig( ) – configure an ATA drive (hard disk or cdrom drive) **321**
ichAtaConfigInit( ) – intialize the hard disk driver **322**
ichAtaCtrlReset( ) – reset the specified ATA/IDE disk controller **322**
ichAtaDevCreate( ) – create a device for a ATA/IDE disk **323**
ichAtaDevIdentify( ) – identify device **324**
ichAtaDmaRW( ) – read/write a number of sectors on the current track in DMA mode **324**
ichAtaDmaToggle( ) – turn on or off an individual controllers dma support **325**
ichAtaDrv( ) – Initialize the ATA driver **325**
ichAtaDumptest( ) – a quick test of the dump functionality for ATA driver **326**
ichAtaInit( ) – initialize ATA device. **327**
ichAtaParamRead( ) – Read drive parameters **327**
ichAtaPiInit( ) – init a ATAPI CD-ROM disk controller **328**
ichAtaRW( ) – read/write a data from/to required sector. **328**
ichAtaRawio( ) – do raw I/O access **329**
ichAtaShow( ) – show the ATA/IDE disk parameters **329**
ichAtaShowInit( ) – initialize the ATA/IDE disk driver show routine **330**
ichAtaStatusChk( ) – Check status of drive and compare to requested status. **330**
ichAtaXbdDevCreate( ) – create an XBD device for a ATA/IDE disk **331**
ichAtaXbdRawio( ) – do raw I/O access **331**
ichAtapiBytesPerSectorGet( ) – get the number of Bytes per sector. **332**
ichAtapiBytesPerTrackGet( ) – get the number of Bytes per track. **332**
ichAtapiCtrlMediumRemoval( ) – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command **333**
ichAtapiCurrentCylinderCountGet( ) – get logical number of cylinders in the drive. **333**
ichAtapiCurrentHeadCountGet( ) – get the number of read/write heads in the drive. **334**
ichAtapiCurrentMDmaModeGet( ) – get the enabled Multi word DMA mode. **334**
ichAtapiCurrentPioModeGet( ) – get the enabled PIO mode. **335**
ichAtapiCurrentRwModeGet( ) – get the current Data transfer mode. **335**
ichAtapiCurrentSDmaModeGet( ) – get the enabled Single word DMA mode. **336**
ichAtapiCurrentUDmaModeGet( ) – get the enabled Ultra DMA mode. **336**
ichAtapiCylinderCountGet( ) – get the number of cylinders in the drive. **337**
ichAtapiDriveSerialNumberGet( ) – get the drive serial number. **338**

ichAtapiDriveTypeGet( ) – get the drive type. **338**
ichAtapiFeatureEnabledGet( ) – get the enabled features. **339**
ichAtapiFeatureSupportedGet( ) – get the features supported by the drive. **340**
ichAtapiFirmwareRevisionGet( ) – get the firm ware revision of the drive. **340**
ichAtapiHeadCountGet( ) – get the number heads in the drive. **341**
ichAtapiInit( ) – init ATAPI CD-ROM disk controller **341**
ichAtapiIoctl( ) – Control the drive. **342**
ichAtapiMaxMDmaModeGet( ) – get the Maximum Multi word DMA mode the drive supports. **346**
ichAtapiMaxPioModeGet( ) – get the Maximum PIO mode that drive can support. **347**
ichAtapiMaxSDmaModeGet( ) – get the Maximum Single word DMA mode the drive supports **347**
ichAtapiMaxUDmaModeGet( ) – get the Maximum Ultra DMA mode the drive can support. **348**
ichAtapiModelNumberGet( ) – get the model number of the drive. **349**
ichAtapiPktCmd( ) – execute an ATAPI command with error processing **349**
ichAtapiPktCmdSend( ) – Issue a Packet command. **350**
ichAtapiRead10( ) – read one or more blocks from an ATAPI Device. **350**
ichAtapiReadCapacity( ) – issue a READ CD-ROM CAPACITY command to a ATAPI device **351**
ichAtapiReadTocPmaAtip( ) – issue a READ TOC command to a ATAPI device **351**
ichAtapiRemovMediaStatusNotifyVerGet( ) – get the Media Stat Notification Version. **352**
ichAtapiScan( ) – issue SCAN packet command to ATAPI drive. **352**
ichAtapiSeek( ) – issues a SEEK packet command to drive. **353**
ichAtapiSetCDSpeed( ) – issue SET CD SPEED packet command to ATAPI drive. **353**
ichAtapiStartStopUnit( ) – Issues START STOP UNIT packet command **354**
ichAtapiStopPlayScan( ) – issue STOP PLAY/SCAN packet command to ATAPI drive. **354**
ichAtapiTestUnitRdy( ) – issue a TEST UNIT READY command to a ATAPI drive **355**
ichAtapiVersionNumberGet( ) – get the ATA/ATAPI version number of the drive. **355**
index( ) – find the first occurrence of a character in a string **356**
infinity( ) – return a very large double **357**
infinityf( ) – return a very large float **357**
inflate( ) – inflate compressed code **358**
intCRGet( ) – read the contents of the cause register (MIPS) **358**
intCRSet( ) – write the contents of the cause register (MIPS) **359**
intConnect( ) – connect a C routine to a hardware interrupt **359**
intContext( ) – determine if executing in interrupt context **362**
intCount( ) – get the current interrupt nesting depth **363**
intCpuLock( ) – lock out interrupts on local CPU **363**
intCpuUnlock( ) – cancel local CPU interrupt lock **364**
intDisable( ) – disable corresponding interrupt bits (MIPS, PowerPC, ARM) **364**
intDisconnect( ) – disconnect a C routine from a hardware interrupt **365**
intEnable( ) – enable corresponding interrupt bits (MIPS, PowerPC, ARM) **366**
intHandlerCreate( ) – construct an interrupt handler for a C routine (MC680x0, x86, MIPS, SimSolaris) **366**
intHandlerCreateI86( ) – construct an interrupt handler for a C routine (x86) **367**
intLevelSet( ) – set the interrupt level (MC680X0, x86, ARM, SimSolaris, SimNT and SH) **368**
intLock( ) – lock out interrupts **369**
intLockLevelGet( ) – get the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT) **371**

intLockLevelSet( ) – set the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT) **372**
intSRGet( ) – read the contents of the status register (MIPS) **372**
intSRSet( ) – update the contents of the status register (MIPS) **373**
intStackEnable( ) – enable or disable the interrupt stack usage (x86) **373**
intUninitVecSet( ) – set the uninitialized vector handler (ARM) **374**
intUnlock( ) – cancel interrupt locks **374**
intVecBaseGet( ) – get the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT) **375**
intVecBaseSet( ) – set the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT) **375**
intVecGet( ) – get an interrupt vector (MC680x0, x86, MIPS, SH, SimSolaris, SimNT) **376**
intVecGet2( ) – get a CPU vector, gate type(int/trap), and gate selector (x86) **377**
intVecSet( ) – set a CPU vector (trap) (MC680x0, x86, MIPS, SH, SimSolaris, SimNT) **378**
intVecSet2( ) – set a CPU vector, gate type(int/trap), and selector (x86) **382**
intVecTableWriteProtect( ) – write-protect exception vector table (MC680x0, x86, ARM, SimSolaris, SimNT) **382**
ioGlobalStdGet( ) – get the file descriptor for global input/output/error **383**
ioGlobalStdSet( ) – set file descriptor for global input/output/error **383**
ioHelp( ) – print a synopsis of I/O utility functions **384**
ioTaskStdGet( ) – get the file descriptor for task standard input/output/error **384**
ioTaskStdSet( ) – set the file descriptor for task standard input/output/error **385**
ioctl( ) – perform an I/O control function **386**
iosDevAdd( ) – add a device to the kernel I/O system **387**
iosDevDelDrv( ) – invoke device delete driver if reference counter reaches 0. **387**
iosDevDelete( ) – delete a device from the kernel I/O system **388**
iosDevFind( ) – find an I/O device in the kernel device list **389**
iosDevShow( ) – display the list of devices in the system **390**
iosDrvInstall( ) – install a kernel I/O driver **390**
iosDrvRemove( ) – remove a kernel I/O driver **391**
iosDrvShow( ) – display a list of system drivers **392**
iosFdEntryGet( ) – get an unused **FD_ENTRY** from the pool **392**
iosFdEntryReturn( ) – return an **FD_ENTRY** to the pool **392**
iosFdMaxFiles( ) – return maximum files for current RTP **393**
iosFdShow( ) – display a list of file descriptor names in the system **393**
iosInit( ) – initialize the kernel I/O system **394**
iosRtpFdShow( ) – show the per-RTP *fd* table **394**
iosShowInit( ) – initialize the I/O system show facility **395**
irint( ) – convert a double-precision value to an integer **395**
irintf( ) – convert a single-precision value to an integer **396**
iround( ) – round a number to the nearest integer **396**
iroundf( ) – round a number to the nearest integer **397**
isatty( ) – return whether the underlying driver is a *tty* device **397**
isrCreate( ) – create an ISR object **398**
isrDelete( ) – delete an ISR object **399**
isrIdSelf( ) – get the ISR ID of the currently running ISR **400**
isrInfoGet( ) – get information about an ISR object **400**

isrInvoke( ) – invoke the handler routine of an ISR object **401**
isrShow( ) – show information about an ISR object **402**
kernelCpuEnable( ) – enable a CPU **403**
kernelInit( ) – initialize the kernel **403**
kernelIsCpuIdle( ) – determine whether the specified CPU is idle **404**
kernelIsSystemIdle( ) – determine whether all enabled processors are idle **405**
kernelRoundRobinInstall( ) – install VxWorks Round Robin implementation **405**
kernelTimeSlice( ) – enable round-robin selection **406**
kernelVersion( ) – return the WIND kernel revision string **407**
kill( ) – send a signal to a task (POSIX) **407**
l( ) – disassemble and display a specified number of instructions **408**
l0( ) – return the contents of register l0 (also l1-l7) (SimSolaris) **409**
ld( ) – load an object module into memory **409**
ledClose( ) – discard the line-editor ID **410**
ledControl( ) – change the line-editor ID parameters **411**
ledLibInit( ) – initialize the line editing facilities **411**
ledOpen( ) – create a new line-editor ID **412**
ledRead( ) – read a line with line-editing **412**
link( ) – link a file **413**
lio_listio( ) – initiate a list of asynchronous I/O requests (POSIX) **413**
lkAddr( ) – list symbols whose values are near a specified value **414**
lkup( ) – list symbols **415**
ll( ) – generate a long listing of directory contents **415**
llr( ) – do a long listing of directory and all its subdirectories contents **416**
lnPciRegister( ) – register with the VxBus subsystem **417**
loadModule( ) – load an object module into memory **417**
loadModuleAt( ) – load an object module into memory **418**
log10f( ) – compute a base-10 logarithm (ANSI) **423**
log2( ) – compute a base-2 logarithm **424**
log2f( ) – compute a base-2 logarithm **424**
logFdAdd( ) – add a logging file descriptor **425**
logFdDelete( ) – delete a logging file descriptor **425**
logFdSet( ) – set the primary logging file descriptor **426**
logInit( ) – initialize message logging library **426**
logMsg( ) – log a formatted error message **427**
logTask( ) – message-logging support task **428**
logf( ) – compute a natural logarithm (ANSI) **428**
loginDefaultEncrypt( ) – default password encryption routine **429**
loginEncryptInstall( ) – install an encryption routine **429**
loginInit( ) – initialize the login table **430**
loginPrompt( ) – display a login prompt and validate a user entry **431**
loginStringSet( ) – change the login string **431**
loginUserAdd( ) – add a user to the login table **432**
loginUserDelete( ) – delete a user entry from the login table **433**
loginUserShow( ) – display the user login table **434**

loginUserVerify( ) – verify a user name and password in the login table **434**
logout( ) – log out of the VxWorks system **435**
ls( ) – generate a brief listing of a directory **435**
lseek( ) – set a file read/write pointer **436**
lsr( ) – list the contents of a directory and any of its subdirectories **437**
lstAdd( ) – add a node to the end of a list **437**
lstConcat( ) – concatenate two lists **438**
lstCount( ) – report the number of nodes in a list **438**
lstDelete( ) – delete a specified node from a list **439**
lstExtract( ) – extract a sublist from a list **439**
lstFind( ) – find a node in a list **440**
lstFirst( ) – find first node in list **440**
lstFree( ) – free up a list **441**
lstGet( ) – delete and return the first node from a list **441**
lstInit( ) – initialize a list descriptor **442**
lstInsert( ) – insert a node in a list after a specified node **442**
lstLast( ) – find the last node in a list **443**
lstNStep( ) – find a list node *nStep* steps away from a specified node **443**
lstNext( ) – find the next node in a list **444**
lstNth( ) – find the Nth node in a list **444**
lstPrevious( ) – find the previous node in a list **445**
m( ) – modify memory **445**
m6845vxbRegister( ) – register m6845vxb driver **446**
m85xxCCSRRegister( ) – register m85xxLAWBAR driver **446**
mRegs( ) – modify registers **447**
mach( ) – return the contents of system register **mach** (also **macl**, **pr**) (SH) **448**
malloc( ) – allocate a block of memory from the system memory partition (ANSI) **448**
memAddToPool( ) – add memory to the system memory partition **449**
memDevCreate( ) – create a memory device **449**
memDevCreateDir( ) – create a memory device for multiple files **451**
memDevDelete( ) – delete a memory device **451**
memDrv( ) – install a memory driver **452**
memEdrBlockMark( ) – mark or unmark selected blocks **452**
memEdrBlockShow( ) – print memory block information **453**
memEdrFreeQueueFlush( ) – flush the free queue **454**
memEdrPartShow( ) – show partition information in the kernel **454**
memEdrRtpBlockMark( ) – mark or unmark selected allocated blocks in an RTP **455**
memEdrRtpBlockShow( ) – print memory block information of an RTP **455**
memEdrRtpPartShow( ) – show partition information of an RTP **456**
memFindMax( ) – find the largest free block in the system memory partition (kernel heap) **457**
memInfoGet( ) – get heap information **457**
memOptionsGet( ) – get the options of the system memory partition (kernel heap) **458**
memOptionsSet( ) – set the options for the system memory partition (kernel heap) **458**
memPartAddToPool( ) – add memory to a memory partition **459**
memPartAlignedAlloc( ) – allocate aligned memory from a partition **460**

memPartAlloc( ) – allocate a block of memory from a partition **460**
memPartCreate( ) – create a memory partition **461**
memPartDelete( ) – delete a partition and free associated memory **462**
memPartFindMax( ) – find the size of the largest available free block **462**
memPartFree( ) – free a block of memory in a partition **463**
memPartInfoGet( ) – get partition information **464**
memPartOptionsGet( ) – get the options of a memory partition **464**
memPartOptionsSet( ) – set the options for a memory partition **465**
memPartRealloc( ) – reallocate a block of memory in a specified partition **466**
memPartShow( ) – show blocks and statistics for a given memory partition **466**
memPartSmCreate( ) – create a shared memory partition (VxMP Option) **467**
memShow( ) – show blocks and statistics for the current heap partition **468**
memShowInit( ) – initialize the memory partition show facility **471**
memalign( ) – allocate aligned memory from system memory partition (kernel heap) **471**
miiBusCreate( ) – create an miiBus attached to a parent bridge **472**
miiBusDelete( ) – delete an miiBus and all its child devices **473**
miiBusGet( ) – get the miiBus that goes with a given VxBus instance **473**
miiBusListAdd( ) – Add a PHY to the MII monitor list **474**
miiBusListDel( ) – Remove a PHY to the MII monitor list **474**
miiBusMediaAdd( ) – add an entry to an miiBus's media list **475**
miiBusMediaDefaultSet( ) – set the default media for an miiBus **475**
miiBusMediaDel( ) – delete an entry to an miiBus's media list **476**
miiBusMediaListGet( ) – obtain a pointer to the bus's media list **476**
miiBusMediaUpdate( ) – invoke a PHY's parent's media update callback **477**
miiBusModeGet( ) – get the current media mode and link status **477**
miiBusModeSet( ) – set the current media mode **478**
miiBusRead( ) – read a PHY register **478**
miiBusRegister( ) – register with the vxBus subsystem **479**
miiBusWrite( ) – write value to a PHY register **479**
mkdir( ) – make a directory **480**
mlock( ) – lock specified pages into memory (POSIX) **480**
mlockall( ) – lock all pages used by a process into memory (POSIX) **481**
mmapShow( ) – show information about memory mapped objects in the system **481**
mmuPhysToVirt( ) – translate a physical address to a virtual address (ARM) **483**
mmuPro32LibInit( ) – initialize module **483**
mmuPro32Page0UnMap( ) – unmap the page zero for **NULL** pointer detection **484**
mmuPro36LibInit( ) – initialize module **484**
mmuPro36Page0UnMap( ) – unmap the page zero for **NULL** pointer detection **485**
mmuPro36PageMap( ) – map 36bit physical memory page to virtual memory page **485**
mmuPro36Translate( ) – translate a virtual address to a 36bit physical address **486**
mmuShLibInit( ) – Initialize the SH MMU library. **486**
mmuVirtToPhys( ) – translate a virtual address to a physical address (ARM) **487**
moduleCheck( ) – verify checksums on all modules loaded in the system **488**
moduleCreate( ) – create and initialize a module **489**
moduleCreateHookAdd( ) – add a routine to be called when a module is added **489**

moduleCreateHookDelete( ) – delete a previously added module create hook routine **490**
moduleDelete( ) – delete module ID information **490**
moduleFindByGroup( ) – find a module by group number **491**
moduleFindByName( ) – find a module by name **492**
moduleFindByNameAndPath( ) – find a module by filename and path **492**
moduleFlagsGet( ) – get the flags associated with a module ID **493**
moduleIdListGet( ) – get a list of loaded modules **493**
moduleInfoGet( ) – get information about an object module **494**
moduleNameGet( ) – get the name associated with a module ID **494**
moduleSegFirst( ) – find the first segment in a module **495**
moduleSegGet( ) – get (delete and return) the first segment from a module **495**
moduleSegNext( ) – find the next segment in a module **496**
moduleShow( ) – show information about loaded modules **497**
mountdInit( ) – initialize the mount daemon **498**
mqPxDescObjIdGet( ) – returns the **OBJ_ID** associated with a mqd_t descriptor **499**
mqPxLibInit( ) – initialize the POSIX message queue library **500**
mqPxShow( ) – display message queue internals **500**
mqPxShowInit( ) – initialize the POSIX message queue show facility **501**
mq_close( ) – close a message queue (POSIX) **501**
mq_getattr( ) – get message queue attributes (POSIX) **502**
mq_notify( ) – notify a task that a message is available on a queue (POSIX) **503**
mq_open( ) – open a message queue (POSIX) **504**
mq_receive( ) – receive a message from a message queue (POSIX) **506**
mq_send( ) – send a message to a message queue (POSIX) **507**
mq_setattr( ) – set message queue attributes (POSIX) **508**
mq_unlink( ) – remove a message queue (POSIX) **509**
msgQClose( ) – close a named message queue **510**
msgQCreate( ) – create and initialize a message queue **511**
msgQDelete( ) – delete a message queue **512**
msgQEvStart( ) – start the event notification process for a message queue **513**
msgQEvStop( ) – stop the event notification process for a message queue **514**
msgQInfoGet( ) – get information about a message queue **515**
msgQInitialize( ) – initialize a pre-allocated message queue **517**
msgQNumMsgs( ) – get the number of messages queued to a message queue **518**
msgQOpen( ) – open a message queue **519**
msgQOpenInit( ) – initialize the message queue open facility **521**
msgQReceive( ) – receive a message from a message queue **521**
msgQSend( ) – send a message to a message queue **523**
msgQShow( ) – show information about a message queue **524**
msgQShowInit( ) – initialize the message queue show facility **525**
msgQSmCreate( ) – create and initialize a shared memory message queue (VxMP Option) **526**
msgQUnlink( ) – unlink a named message queue **527**
munlock( ) – unlock specified pages (POSIX) **528**
munlockall( ) – unlock all pages used by a process (POSIX) **528**
mv( ) – mv file into other directory. **529**

nanosleep( ) – suspend the current task until the time interval elapses (POSIX) **530**
netHelp( ) – print a synopsis of network routines **530**
nfsAuthUnixGet( ) – get the NFS UNIX authentication parameters **531**
nfsAuthUnixPrompt( ) – modify the NFS UNIX authentication parameters **532**
nfsAuthUnixSet( ) – set the NFS UNIX authentication parameters **533**
nfsAuthUnixShow( ) – display the NFS UNIX authentication parameters **533**
nfsChkFilePerms( ) – check the NFS file permissions with a given permission. **534**
nfsDevInfoGet( ) – read configuration information from the requested device **534**
nfsDevListGet( ) – create list of all the NFS devices in the system **535**
nfsDevShow( ) – display the mounted NFS devices **535**
nfsDrvNumGet( ) – Get driver number of NFS device **536**
nfsErrnoSet( ) – set NFS status **536**
nfsExport( ) – specify a file system to be NFS exported **537**
nfsExportShow( ) – display the exported file systems of a remote host **538**
nfsHelp( ) – display the NFS help menu **538**
nfsIdSet( ) – set the ID number of the NFS UNIX authentication parameters **539**
nfsMntDump( ) – display all NFS file systems mounted on a particular host **540**
nfsMount( ) – mount an NFS file system **540**
nfsMountAll( ) – mount all file systems exported by a specified host **541**
nfsStatusGet( ) – Get the statistics of the NFS server **542**
nfsUnexport( ) – remove a file system from the list of exported file systems **542**
nfsUnmount( ) – unmount an NFS device **543**
nfsdHashTableParamsSet( ) – sets up the parameters for the NFS hash table **543**
nfsdInit( ) – initialize the NFS server **544**
nfsdStatusShow( ) – show the status of the NFS server **545**
nicRegister( ) – register with the VxBus subsystem **546**
npc( ) – return the contents of the next program counter (SimSolaris) **546**
nseRegister( ) – register with the VxBus subsystem **547**
nvRamSegDefGet( ) – get segment allocation from BSP **547**
o0( ) – return the contents of register o0 (also o1-o7) (SimSolaris) **548**
objClassTypeGet( ) – get an object's class type **549**
objContextGet( ) – return the object's context value **549**
objContextSet( ) – set the object's context value **550**
objHandleShow( ) – show information on the object referenced by an object handle **550**
objHandleTblShow( ) – show information on an RTP's handle table **551**
objNameGet( ) – get an object's name **551**
objNameLenGet( ) – get an object's name length **552**
objNameToId( ) – find object with matching name string and type **552**
objOwnerGet( ) – return the object's owner **554**
objOwnerSet( ) – change the object's owner **554**
objShow( ) – show information on an object **555**
objShowAll( ) – show all information on an object **555**
open( ) – open a file **557**
opendir( ) – open a directory for searching (POSIX) **559**
operator_delete( ) – default run-time support for memory deallocation (C++) **560**

operator_new( ) – default run-time support for operator new (C++) **560**
operator_new( ) – default run-time support for operator new (nothrow) (C++) **561**
operator_new( ) – run-time support for operator new with placement (C++) **561**
oprintf( ) – write a formatted string to an output function **562**
partLibCreate( ) – partition a device **563**
passFsDevInit( ) – associate a device with passFs file system functions **563**
passFsInit( ) – prepare to use the passFs library **564**
pathconf( ) – determine the current value of a configurable limit **564**
pause( ) – suspend the task until delivery of a signal (POSIX) **565**
pc( ) – return the contents of the program counter **565**
pcConDevBind( ) – bind keyboard or VGA device with console **566**
pcConDevCreate( ) – create a device for the on-board ports **566**
pcConDrv( ) – initialize the console driver **567**
pentiumBtc( ) – execute atomic compare-and-exchange instruction to clear a bit **567**
pentiumBts( ) – execute atomic compare-and-exchange instruction to set a bit **568**
pentiumCr4Get( ) – get contents of CR4 register **568**
pentiumCr4Set( ) – sets specified value to the CR4 register **568**
pentiumMcaEnable( ) – enable/disable the MCA (Machine Check Architecture) **569**
pentiumMcaShow( ) – show MCA (Machine Check Architecture) registers **569**
pentiumMsrGet( ) – get the contents of the specified MSR (Model Specific Register) **570**
pentiumMsrInit( ) – initialize all the MSRs (Model Specific Register) **570**
pentiumMsrSet( ) – set a value to the specified MSR (Model Specific Registers) **571**
pentiumMsrShow( ) – show all the MSR (Model Specific Register) **571**
pentiumMtrrDisable( ) – disable MTRR (Memory Type Range Register) **572**
pentiumMtrrEnable( ) – enable MTRR (Memory Type Range Register) **572**
pentiumMtrrGet( ) – get MTRRs to a specified MTRR table **572**
pentiumMtrrSet( ) – set MTRRs from specified MTRR table with WRMSR instruction. **573**
pentiumP5PmcGet( ) – get the contents of P5 PMC0 and PMC1 **573**
pentiumP5PmcGet0( ) – get the contents of P5 PMC0 **574**
pentiumP5PmcGet1( ) – get the contents of P5 PMC1 **574**
pentiumP5PmcReset( ) – reset both PMC0 and PMC1 **575**
pentiumP5PmcReset0( ) – reset PMC0 **575**
pentiumP5PmcReset1( ) – reset PMC1 **575**
pentiumP5PmcStart0( ) – start PMC0 **576**
pentiumP5PmcStart1( ) – start PMC1 **576**
pentiumP5PmcStop( ) – stop both P5 PMC0 and PMC1 **577**
pentiumP5PmcStop0( ) – stop P5 PMC0 **577**
pentiumP5PmcStop1( ) – stop P5 PMC1 **577**
pentiumP6PmcGet( ) – get the contents of PMC0 and PMC1 **578**
pentiumP6PmcGet0( ) – get the contents of PMC0 **578**
pentiumP6PmcGet1( ) – get the contents of PMC1 **579**
pentiumP6PmcReset( ) – reset both PMC0 and PMC1 **579**
pentiumP6PmcReset0( ) – reset PMC0 **579**
pentiumP6PmcReset1( ) – reset PMC1 **580**
pentiumP6PmcStart( ) – start both PMC0 and PMC1 **580**

pentiumP6PmcStop( ) – stop both PMC0 and PMC1 **581**
pentiumP6PmcStop1( ) – stop PMC1 **581**
pentiumPmcGet( ) – get the contents of PMC0 and PMC1 **581**
pentiumPmcGet0( ) – get the contents of PMC0 **582**
pentiumPmcGet1( ) – get the contents of PMC1 **582**
pentiumPmcReset( ) – reset both PMC0 and PMC1 **583**
pentiumPmcReset0( ) – reset PMC0 **583**
pentiumPmcReset1( ) – reset PMC1 **583**
pentiumPmcShow( ) – show PMCs (Performance Monitoring Counters) **584**
pentiumPmcStart( ) – start both PMC0 and PMC1 **584**
pentiumPmcStart0( ) – start PMC0 **585**
pentiumPmcStart1( ) – start PMC1 **585**
pentiumPmcStop( ) – stop both PMC0 and PMC1 **585**
pentiumPmcStop0( ) – stop PMC0 **586**
pentiumPmcStop1( ) – stop PMC1 **586**
pentiumSerialize( ) – execute a serializing instruction CPUID **587**
pentiumTlbFlush( ) – flush TLBs (Translation Lookaside Buffers) **587**
pentiumTscGet32( ) – get the lower half of the 64Bit TSC (Timestamp Counter) **588**
pentiumTscGet64( ) – get 64Bit TSC (Timestamp Counter) **588**
pentiumTscReset( ) – reset the TSC (Timestamp Counter) **588**
period( ) – spawn a task to call a function periodically **589**
periodRun( ) – call a function periodically **590**
philDemo( ) – entry point for VxWorks/SMP Dijkstra's dining philosophers demo **590**
pipeDevCreate( ) – create a pipe device **591**
pipeDevDelete( ) – delete a pipe device **591**
pipeDrv( ) – initialize the pipe driver **592**
pmFreeSpace( ) – returns the amount of free space left in the PM arena **593**
pmInvalidate( ) – invalidates the entire PM arena **593**
pmRegionAddr( ) – returns the address of a persistent heap region **594**
pmRegionClose( ) – closes a region making it inaccessible to clients **594**
pmRegionCreate( ) – creates a persistent heap region **595**
pmRegionOpen( ) – opens an existing persistent heap region **595**
pmRegionProtect( ) – makes a PM region read-only **596**
pmRegionSize( ) – return the size of a persistent heap region **596**
pmShow( ) – shows the created persistent heap segments **597**
pmValidate( ) – validates a PM arena **597**
poolBlockAdd( ) – add an item block to the pool **598**
poolCreate( ) – create a pool **598**
poolDelete( ) – delete a pool **599**
poolFreeCount( ) – return number of free items in pool **600**
poolIncrementGet( ) – get the increment value used to grow the pool **600**
poolIncrementSet( ) – set the increment value used to grow the pool **601**
poolItemGet( ) – get next free item from pool and return a pointer to it **601**
poolItemReturn( ) – return an item to the pool **602**
poolShow( ) – display pool information **603**

poolTotalCount( ) – return total number of items in pool **604**
poolUnusedBlocksFree( ) – free blocks that have all items unused **604**
powf( ) – compute the value of a number raised to a specified power (ANSI) **605**
primesCompute( ) – entry point for the VxWorks SMP prime number computation demo **606**
printErr( ) – write a formatted string to the standard error stream **606**
printErrno( ) – print the definition of a specified error status value **607**
printLogo( ) – print the VxWorks logo **607**
printf( ) – write a formatted string to the standard output stream (ANSI) **608**
proofUtf8( ) – Determine if a string represents a valid UTF-8 character **612**
proofUtf8String( ) – determine if a string is valid UTF-8 **612**
psr( ) – return the contents of the processor status register (SimSolaris) **613**
psrShow( ) – display the meaning of a specified PSR value, symbolically (ARM) **613**
pthread_attr_destroy( ) – destroy a thread attributes object (POSIX) **614**
pthread_attr_getdetachstate( ) – get value of detachstate attribute from thread attributes object (POSIX) **614**
pthread_attr_getinheritsched( ) – get current value if inheritsched attribute in thread attributes object (POSIX) **615**
pthread_attr_getname( ) – get name of thread attribute object **615**
pthread_attr_getopt( ) – get options from thread attribute object **616**
pthread_attr_getschedparam( ) – get value of schedparam attribute from thread attributes object (POSIX) **616**
pthread_attr_getschedpolicy( ) – get schedpolicy attribute from thread attributes object (POSIX) **617**
pthread_attr_getscope( ) – get contention scope from thread attributes (POSIX) **617**
pthread_attr_getstackaddr( ) – get value of stackaddr attribute from thread attributes object (POSIX) **618**
pthread_attr_getstacksize( ) – get stack value of stacksize attribute from thread attributes object (POSIX) **618**
pthread_attr_init( ) – initialize thread attributes object (POSIX) **619**
pthread_attr_setdetachstate( ) – set detachstate attribute in thread attributes object (POSIX) **620**
pthread_attr_setinheritsched( ) – set inheritsched attribute in thread attribute object (POSIX) **621**
pthread_attr_setname( ) – set name in thread attribute object **621**
pthread_attr_setopt( ) – set options in thread attribute object **622**
pthread_attr_setschedparam( ) – set schedparam attribute in thread attributes object (POSIX) **622**
pthread_attr_setschedpolicy( ) – set schedpolicy attribute in thread attributes object (POSIX) **623**
pthread_attr_setscope( ) – set contention scope for thread attributes (POSIX) **624**
pthread_attr_setstackaddr( ) – set stackaddr attribute in thread attributes object (POSIX) **624**
pthread_attr_setstacksize( ) – set stacksize attribute in thread attributes object (POSIX) **625**
pthread_cancel( ) – cancel execution of a thread (POSIX) **626**
pthread_cleanup_pop( ) – pop a cleanup routine off the top of the stack (POSIX) **626**
pthread_cleanup_push( ) – pushes a routine onto the cleanup stack (POSIX) **627**
pthread_cond_broadcast( ) – unblock all threads waiting on a condition (POSIX) **628**
pthread_cond_destroy( ) – destroy a condition variable (POSIX) **628**
pthread_cond_init( ) – initialize condition variable (POSIX) **629**
pthread_cond_signal( ) – unblock a thread waiting on a condition (POSIX) **629**
pthread_cond_timedwait( ) – wait for a condition variable with a timeout (POSIX) **630**
pthread_cond_wait( ) – wait for a condition variable (POSIX) **631**
pthread_condattr_destroy( ) – destroy a condition attributes object (POSIX) **632**
pthread_condattr_init( ) – initialize a condition attribute object (POSIX) **632**

pthread_create( ) – create a thread (POSIX) **633**
pthread_detach( ) – dynamically detach a thread (POSIX) **633**
pthread_equal( ) – compare thread IDs (POSIX) **634**
pthread_exit( ) – terminate a thread (POSIX) **634**
pthread_getschedparam( ) – get value of schedparam attribute from a thread (POSIX) **635**
pthread_getspecific( ) – get thread specific data (POSIX) **636**
pthread_join( ) – wait for a thread to terminate (POSIX) **636**
pthread_key_create( ) – create a thread specific data key (POSIX) **637**
pthread_key_delete( ) – delete a thread specific data key (POSIX) **638**
pthread_kill( ) – send a signal to a thread (POSIX) **638**
pthread_mutex_destroy( ) – destroy a mutex (POSIX) **639**
pthread_mutex_getprioceiling( ) – get the value of the prioceiling attribute of a mutex (POSIX) **639**
pthread_mutex_init( ) – initialize mutex from attributes object (POSIX) **640**
pthread_mutex_lock( ) – lock a mutex (POSIX) **641**
pthread_mutex_setprioceiling( ) – dynamically set the prioceiling attribute of a mutex (POSIX) **641**
pthread_mutex_trylock( ) – lock mutex if it is available (POSIX) **642**
pthread_mutex_unlock( ) – unlock a mutex (POSIX) **643**
pthread_mutexattr_destroy( ) – destroy mutex attributes object (POSIX) **643**
pthread_mutexattr_getprioceiling( ) – get the current value of the prioceiling attribute in a mutex attributes object (POSIX) **644**
pthread_mutexattr_getprotocol( ) – get value of protocol in mutex attributes object (POSIX) **644**
pthread_mutexattr_init( ) – initialize mutex attributes object (POSIX) **645**
pthread_mutexattr_setprioceiling( ) – set prioceiling attribute in mutex attributes object (POSIX) **645**
pthread_mutexattr_setprotocol( ) – set protocol attribute in mutex attribute object (POSIX) **646**
pthread_once( ) – dynamic package initialization (POSIX) **647**
pthread_self( ) – get the calling thread's ID (POSIX) **648**
pthread_setcancelstate( ) – set cancellation state for calling thread (POSIX) **648**
pthread_setcanceltype( ) – set cancellation type for calling thread (POSIX) **649**
pthread_setschedparam( ) – dynamically set schedparam attribute for a thread (POSIX) **649**
pthread_setspecific( ) – set thread specific data (POSIX) **650**
pthread_sigmask( ) – change and/or examine calling thread's signal mask (POSIX) **650**
pthread_testcancel( ) – create a cancellation point in the calling thread (POSIX) **651**
ptyDevCreate( ) – create a pseudo terminal **652**
ptyDevRemove( ) – destroy a pseudo terminal **652**
ptyDrv( ) – initialize the pseudo-terminal driver **653**
putenv( ) – set an environment variable **653**
pwd( ) – print the current default directory **654**
quiccEngineDrvCtrlShow( ) – place holder just prints out control structure ptr **654**
quiccEngineRegister( ) – register quiccEngine driver **655**
r0( ) – return the contents of register **r0** (also **r1** - **r14**) (ARM) **655**
r0( ) – return the contents of general register **r0** (also **r1**-`r15') (SH) **656**
raise( ) – send a signal to the caller's task **656**
ramDevCreate( ) – create a RAM disk device **657**
ramDiskDevCreate( ) – Initialize a RAM Disk device **658**
ramDrv( ) – prepare a RAM disk driver for use (optional) **659**

rawFsDevInit( ) – associate a block device with raw volume functions **660**
rawFsInit( ) – prepare to use the raw volume library **660**
rawPerfDemo( ) – entry point for the VxWorks/SMP raw performance demo **661**
read( ) – read bytes from a file or device **661**
readdir( ) – read one entry from a directory (POSIX) **662**
readdir_r( ) – read one entry from a directory (POSIX) **663**
realloc( ) – reallocate a block of memory (ANSI) **664**
reboot( ) – reset network devices and transfer control to boot ROMs **665**
rebootHookAdd( ) – add a routine to be called at reboot **665**
reld( ) – reload an object module (shell command) **666**
rename( ) – change the name of a file **667**
repeat( ) – spawn a task to call a function repeatedly **668**
repeatRun( ) – call a function repeatedly **668**
rewinddir( ) – reset position to the start of a directory (POSIX) **669**
rindex( ) – find the last occurrence of a character in a string **670**
rm( ) – remove a file **670**
rmdir( ) – remove a directory **671**
rngBufGet( ) – get characters from a ring buffer **671**
rngBufPut( ) – put bytes into a ring buffer **672**
rngCreate( ) – create an empty ring buffer **672**
rngDelete( ) – delete a ring buffer **673**
rngFlush( ) – make a ring buffer empty **673**
rngFreeBytes( ) – determine the number of free bytes in a ring buffer **674**
rngIsEmpty( ) – test if a ring buffer is empty **674**
rngIsFull( ) – test if a ring buffer is full (no more room) **674**
rngMoveAhead( ) – advance a ring pointer by *n* bytes **675**
rngNBytes( ) – determine the number of bytes in a ring buffer **675**
rngPutAhead( ) – put a byte ahead in a ring buffer without moving ring pointers **676**
romStart( ) – generic ROM initialization **676**
round( ) – round a number to the nearest integer **677**
roundf( ) – round a number to the nearest integer **677**
rtgRegister( ) – register with the VxBus subsystem **678**
rtlRegister( ) – register with the VxBus subsystem **678**
rtpDelete( ) – terminates a real time process (RTP) **679**
rtpDeleteHookAdd( ) – add a routine to be called when RTPs are deleted **680**
rtpDeleteHookDelete( ) – delete a previously added RTP delete hook routine **681**
rtpHelp( ) – print a synopsis of RTP-related shell commands **681**
rtpHookShow( ) – display all installed RTP hooks **682**
rtpInfoGet( ) – Get specific information on an RTP **683**
rtpInitCompleteHookAdd( ) – Add routine to be called after RTP init-complete. **684**
rtpInitCompleteHookDelete( ) – delete a previously added RTP init-complete hook **685**
rtpKill( ) – send a signal to a RTP **685**
rtpLkAddr( ) – list symbols in an RTP whose values are near a specified value **686**
rtpLkup( ) – list symbols from an RTP's symbol table **686**
rtpMemShow( ) – display memory context information for real time proceses **687**

rtpPostCreateHookAdd( ) – add a routine to be called just after RTP creation. **690**
rtpPostCreateHookDelete( ) – delete a previously added RTP post-create hook. **691**
rtpPreCreateHookAdd( ) – add a routine to be called before RTP creation. **691**
rtpPreCreateHookDelete( ) – delete a previously added RTP pre-create hook. **692**
rtpShlShow( ) – Display shared library information for an RTP **692**
rtpShow( ) – display information for real time proceses **693**
rtpSigqueue( ) – send a queued signal to a RTP **696**
rtpSp( ) – launch a RTP with default options. **697**
rtpSpawn( ) – spawns a new Real Time Process (RTP) in the system **698**
rtpSymTblIdGet( ) – Get the symbol table ID of an RTP **702**
rtpSymsAdd( ) – add symbols from an executable file to a RTP symbol table **703**
rtpSymsOverride( ) – override the RTP symbol registration policy **704**
rtpSymsRemove( ) – remove symbols from a RTP symbol table **705**
rtpTaskKill( ) – send a signal to a task **705**
rtpTaskSigqueue( ) – send a queued signal to a task **706**
rtpi( ) – display all tasks within an RTP **706**
s( ) – single-step a task **707**
salCall( ) – invoke a socket-based server **708**
salCreate( ) – create a named socket-based server **708**
salDelete( ) – delete a named socket-based server **710**
salNameFind( ) – find services with the specified name **711**
salOpen( ) – establish communication with a named socket-based server **712**
salRemove( ) – Remove service from SNS by name **713**
salRun( ) – activate a socket-based server **714**
salServerRtnSet( ) – configures the processing routine with the SAL server **715**
salSocketFind( ) – find sockets for a named socket-based server **715**
sbeRegister( ) – register with the VxBus subsystem **717**
scMemValEnable( ) – enable or disable pointer/buffer validation in system calls **717**
scMemValidate( ) – validate an address range passed to a system call routine **718**
sched_get_priority_max( ) – get the maximum priority (POSIX) **719**
sched_get_priority_min( ) – get the minimum priority (POSIX) **720**
sched_getparam( ) – get the scheduling parameters for a specified task (POSIX) **720**
sched_getscheduler( ) – get the current scheduling policy (POSIX) **721**
sched_rr_get_interval( ) – get the current time slice (POSIX) **721**
sched_setparam( ) – set a task's priority (POSIX) **722**
sched_setscheduler( ) – set scheduling policy and scheduling parameters (POSIX) **723**
sched_yield( ) – relinquish the CPU (POSIX) **723**
scsi2IfInit( ) – initialize the SCSI-2 interface to **scsiLib** **724**
scsiAutoConfig( ) – configure all devices connected to a SCSI controller **724**
scsiBlkDevCreate( ) – define a logical partition on a SCSI block device **725**
scsiBlkDevInit( ) – initialize fields in a SCSI logical partition **725**
scsiBlkDevShow( ) – show the **BLK_DEV** structures on a specified physical device **726**
scsiBusReset( ) – pulse the reset signal on the SCSI bus **726**
scsiCacheSnoopDisable( ) – inform SCSI that hardware snooping of caches is disabled **727**
scsiCacheSnoopEnable( ) – inform SCSI that hardware snooping of caches is enabled **727**

scsiCacheSynchronize( ) – synchronize the caches for data coherency **728**
scsiErase( ) – issue an ERASE command to a SCSI device **729**
scsiFormatUnit( ) – issue a **FORMAT_UNIT** command to a SCSI device **729**
scsiIdentMsgBuild( ) – build an identification message **730**
scsiIdentMsgParse( ) – parse an identification message **731**
scsiInquiry( ) – issue an INQUIRY command to a SCSI device **731**
scsiIoctl( ) – perform a device-specific I/O control function **732**
scsiLoadUnit( ) – issue a LOAD/UNLOAD command to a SCSI device **732**
scsiMgrBusReset( ) – handle a controller-bus reset event **733**
scsiMgrCtrlEvent( ) – send an event to the SCSI controller state machine **733**
scsiMgrEventNotify( ) – notify the SCSI manager of a SCSI (controller) event **734**
scsiMgrShow( ) – show status information for the SCSI manager **735**
scsiMgrThreadEvent( ) – send an event to the thread state machine **735**
scsiModeSelect( ) – issue a **MODE_SELECT** command to a SCSI device **736**
scsiModeSense( ) – issue a **MODE_SENSE** command to a SCSI device **737**
scsiMsgInComplete( ) – handle a complete SCSI message received from the target **737**
scsiMsgOutComplete( ) – perform post-processing after a SCSI message is sent **738**
scsiMsgOutReject( ) – perform post-processing when an outgoing message is rejected **738**
scsiPhysDevCreate( ) – create a SCSI physical device structure **739**
scsiPhysDevDelete( ) – delete a SCSI physical-device structure **740**
scsiPhysDevIdGet( ) – return a pointer to a **SCSI_PHYS_DEV** structure **740**
scsiPhysDevShow( ) – show status information for a physical device **741**
scsiRdSecs( ) – read sector(s) from a SCSI block device **741**
scsiRdTape( ) – read bytes or blocks from a SCSI tape device **742**
scsiReadCapacity( ) – issue a **READ_CAPACITY** command to a SCSI device **742**
scsiRelease( ) – issue a RELEASE command to a SCSI device **743**
scsiReleaseUnit( ) – issue a RELEASE UNIT command to a SCSI device **743**
scsiReqSense( ) – issue a **REQUEST_SENSE** command to a SCSI device and read results **743**
scsiReserve( ) – issue a RESERVE command to a SCSI device **744**
scsiReserveUnit( ) – issue a RESERVE UNIT command to a SCSI device **744**
scsiRewind( ) – issue a REWIND command to a SCSI device **745**
scsiSeqDevCreate( ) – create a SCSI sequential device **745**
scsiSeqIoctl( ) – perform an I/O control function for sequential access devices **746**
scsiSeqReadBlockLimits( ) – issue a **READ_BLOCK_LIMITS** command to a SCSI device **747**
scsiSeqStatusCheck( ) – detect a change in media **747**
scsiShow( ) – list the physical devices attached to a SCSI controller **748**
scsiSpace( ) – move the tape on a specified physical SCSI device **748**
scsiStartStopUnit( ) – issue a **START_STOP_UNIT** command to a SCSI device **749**
scsiSyncXferNegotiate( ) – initiate or continue negotiating transfer parameters **749**
scsiTapeModeSelect( ) – issue a **MODE_SELECT** command to a SCSI tape device **750**
scsiTapeModeSense( ) – issue a **MODE_SENSE** command to a SCSI tape device **751**
scsiTargetOptionsGet( ) – get options for one or all SCSI targets **751**
scsiTargetOptionsSet( ) – set options for one or all SCSI targets **752**
scsiTargetOptionsShow( ) – display options for specified SCSI target **753**
scsiTestUnitRdy( ) – issue a **TEST_UNIT_READY** command to a SCSI device **753**

scsiThreadInit( ) – perform generic SCSI thread initialization **754**
scsiWideXferNegotiate( ) – initiate or continue negotiating wide parameters **754**
scsiWrtFileMarks( ) – write file marks to a SCSI sequential device **755**
scsiWrtSecs( ) – write sector(s) to a SCSI block device **755**
scsiWrtTape( ) – write data to a SCSI tape device **756**
sdCreate( ) – create a new shared data region **757**
sdCreateHookAdd( ) – add a hook routine to be called at Shared Data creation **759**
sdCreateHookDelete( ) – delete a Shared Data creation hook routine **760**
sdDelete( ) – delete a shared data region **760**
sdDeleteHookAdd( ) – add a hook routine to be called at Shared Data deletion **761**
sdDeleteHookDelete( ) – delete a Shared Data deletion hook routine **761**
sdGenericHookAdd( ) – add a hook routine to be called before Shared Data routine **762**
sdGenericHookDelete( ) – delete a Shared Data generic hook routine **763**
sdInfoGet( ) – get specific information about a Shared Data Region **764**
sdMap( ) – map a shared data region into an application or the kernel **765**
sdOpen( ) – open a shared data region for use **766**
sdProtect( ) – change the protection attributes of a mapped SD **768**
sdShow( ) – display information for shared data regions **770**
sdUnmap( ) – unmap a shared data region from an application or the kernel **772**
selNodeAdd( ) – add a wake-up node to a **select( )** wake-up list **772**
selNodeDelete( ) – find and delete a node from a **select( )** wake-up list **773**
selWakeup( ) – wake up a task pended in **select( )** **773**
selWakeupAll( ) – wake up all tasks in a **select( )** wake-up list **774**
selWakeupListInit( ) – initialize a **select( )** wake-up list **774**
selWakeupListLen( ) – get the number of nodes in a **select( )** wake-up list **775**
selWakeupListTerm( ) – terminate a **select( )** wake-up list **775**
selWakeupType( ) – get the type of a **select( )** wake-up node **776**
select( ) – pend on a set of file descriptors **776**
selectInit( ) – initialize the select facility **778**
semBCreate( ) – create and initialize a binary semaphore **778**
semBInitialize( ) – initialize a pre-allocated binary semaphore. **779**
semBSmCreate( ) – create and initialize a shared memory binary semaphore (VxMP Option) **780**
semCCreate( ) – create and initialize a counting semaphore **781**
semCInitialize( ) – initialize a pre-allocated counting semaphore. **782**
semCSmCreate( ) – create and initialize a shared memory counting semaphore (VxMP Option) **783**
semClose( ) – close a named semaphore **784**
semDelete( ) – delete a semaphore **785**
semEvStart( ) – start the event notification process for a semaphore **786**
semEvStop( ) – stop the event notification process for a semaphore **788**
semExchange( ) – atomically give and take a pair of semaphores **788**
semFlush( ) – unblock every task pended on a semaphore **790**
semGive( ) – give a semaphore **791**
semInfo( ) – get information about tasks blocked on a semaphore **792**
semInfoGet( ) – get information about a semaphore **792**
semMCreate( ) – create and initialize a mutual-exclusion semaphore **794**

semMGiveForce( ) – give a mutual-exclusion semaphore without restrictions **795**
semMInitialize( ) – initialize a pre-allocated mutex semaphore. **796**
semOpen( ) – open a named semaphore **797**
semOpenInit( ) – initialize the semaphore open facility **799**
semPxLibInit( ) – initialize POSIX semaphore support **799**
semPxShow( ) – display semaphore internals **800**
semPxShowInit( ) – initialize the POSIX semaphore show facility **800**
semRTake( ) – take a semaphore as a reader **801**
semRWCreate( ) – create and initialize a reader/writer semaphore **802**
semRWGiveForce( ) – give a reader/writer semaphore without restrictions **803**
semRWInitialize( ) – initialize a pre-allocated read/write semaphore. **804**
semShow( ) – show information about a semaphore **805**
semTake( ) – take a semaphore **806**
semUnlink( ) – unlink a named semaphore **807**
semWTake( ) – take a semaphore in write mode **808**
sem_close( ) – close a named semaphore (POSIX) **809**
sem_destroy( ) – destroy an unnamed semaphore (POSIX) **810**
sem_getvalue( ) – get the value of a semaphore (POSIX) **810**
sem_init( ) – initialize an unnamed semaphore (POSIX) **811**
sem_open( ) – initialize/open a named semaphore (POSIX) **812**
sem_post( ) – unlock (give) a semaphore (POSIX) **813**
sem_timedwait( ) – lock (take) a semaphore with a timeout (POSIX) **814**
sem_trywait( ) – lock (take) a semaphore, returning error if unavailable (POSIX) **815**
sem_unlink( ) – remove a named semaphore (POSIX) **816**
sem_wait( ) – lock (take) a semaphore, blocking if not available (POSIX) **817**
set_new_handler( ) – set new_handler to user-defined function (C++) **817**
set_terminate( ) – set terminate to user-defined function (C++) **818**
shConfig( ) – display or set the shell configuration **818**
shellAbort( ) – abort a shell session **819**
shellCmdAdd( ) – add a shell command **819**
shellCmdAliasAdd( ) – add an alias string **821**
shellCmdAliasArrayAdd( ) – add an array of alias strings **822**
shellCmdAliasDelete( ) – delete an alias **822**
shellCmdArrayAdd( ) – add an array of shell commands **823**
shellCmdExec( ) – execute a shell command **823**
shellCmdMemRegister( ) – register a buffer against the command interpreter **824**
shellCmdMemUnregister( ) – unregister a buffer **825**
shellCmdPreParseAdd( ) – define a command to be pre-parsed **825**
shellCmdSymTabIdGet( ) – get symbol table Id of a shell session **826**
shellCmdTopicAdd( ) – add a shell command topic **827**
shellCompatibleCheck( ) – check the compatibility mode of the shell **827**
shellConfigDefaultGet( ) – get default shell configuration **828**
shellConfigDefaultSet( ) – set default shell configuration **828**
shellConfigDefaultValueGet( ) – get a default configuration variable value **829**
shellConfigDefaultValueSet( ) – set a default configuration variable value **829**

shellConfigDefaultValueUnset( ) – unset a default configuration variable value **830**
shellConfigGet( ) – get the shell configuration **830**
shellConfigSet( ) – set shell configuration **831**
shellConfigValueGet( ) – get a shell configuration variable value **831**
shellConfigValueSet( ) – set a shell configuration variable value **832**
shellConfigValueUnset( ) – unset a shell configuration variable value **833**
shellDataAdd( ) – add user data to a specified shell **833**
shellDataFirst( ) – get the first user data that matchs a key **834**
shellDataFromNameAdd( ) – add user data to a specified shell **835**
shellDataFromNameGet( ) – get user data from a specified shell **835**
shellDataGet( ) – get user data from a specified shell **836**
shellDataNext( ) – get the next user data that matchs a key **837**
shellDataRemove( ) – remove user data from a specified shell **837**
shellErrnoGet( ) – get the shell session errno **838**
shellErrnoSet( ) – set the shell session errno **838**
shellFirst( ) – get the first shell session **839**
shellFromNameGet( ) – get a shell session Id from a task name **839**
shellFromTaskGet( ) – get a shell session Id from its task Id **839**
shellGenericInit( ) – start a shell session **840**
shellHistory( ) – display or set the size of the shell history (vxWorks 5.5 compatibility) **841**
shellIdVerify( ) – verify the validity of a shell session Id **842**
shellInterpByNameFind( ) – Find an interpreter based on its name **842**
shellInterpCtxGet( ) – get the interpreter context **843**
shellInterpDefaultNameGet( ) – get the name of the default interpreter **843**
shellInterpEvaluate( ) – interpret a string by an interpreter **844**
shellInterpNameGet( ) – get the name of the current interpreter **844**
shellInterpRegister( ) – register a new interpreter **845**
shellLock( ) – lock access to the shell (vxWorks 5.5 compatibility) **847**
shellNext( ) – get the next shell session **848**
shellPromptFmtDftSet( ) – set the default prompt format string **848**
shellPromptFmtSet( ) – set the current prompt format string **849**
shellPromptFmtStrAdd( ) – add a new prompt format string **850**
shellPromptSet( ) – change the shell prompt (vxWorks 5.5 compatibility) **850**
shellResourceReleaseHookAdd( ) – add a resource-releasing hook to the shell **851**
shellRestart( ) – restart a shell session **852**
shellScriptAbort( ) – signal the shell to stop processing a script (vxWorks 5.5 compatibility) **852**
shellTaskGet( ) – get the task Id of a shell session **853**
shellTaskIdDefault( ) – set the default task for a given shell session **853**
shellTerminate( ) – terminate a shell task **854**
shlShow( ) – display information for shared libraries **855**
shlSymsAdd( ) – add symbols from a shared object file to a RTP symbol table **856**
shlSymsRemove( ) – remove shared library symbols from a RTP symbol table **857**
show( ) – print information on a specified object **858**
sigInit( ) – initialize the signal facilities **858**
sigaction( ) – examine and/or specify the action associated with a signal (POSIX) **859**

sigaddset( ) – add a signal to a signal set (POSIX) **859**
sigblock( ) – add to a set of blocked signals **860**
sigdelset( ) – delete a signal from a signal set (POSIX) **860**
sigemptyset( ) – initialize a signal set with no signals included (POSIX) **861**
sigfillset( ) – initialize a signal set with all signals included (POSIX) **861**
sigismember( ) – test to see if a signal is in a signal set (POSIX) **861**
signal( ) – specify the handler associated with a signal **862**
sigpending( ) – retrieve the set of pending signals blocked from delivery (POSIX) **863**
sigprocmask( ) – examine and/or change the signal mask (POSIX) **863**
sigqueue( ) – send a queued signal to a task **864**
sigqueueInit( ) – initialize the queued signal facilities **864**
sigsetmask( ) – set the signal mask **865**
sigsuspend( ) – suspend the task until delivery of a signal (POSIX) **865**
sigtimedwait( ) – wait for a signal **866**
sigvec( ) – install a signal handler **867**
sigwait( ) – wait for a signal to be delivered (POSIX) **868**
sigwaitinfo( ) – wait for real-time signals **868**
sil31xxBIST( ) – Controller Built-In Self Test... **869**
sil31xxBISTShow( ) – Show the results of the power-on BIST **869**
sil31xxDiskPresent( ) – Return **OK** if disk exists. **870**
sil31xxDrvVxbInit( ) – Initialize the driver. **870**
sil31xxIsr( ) – Interrupt service routine. **871**
sil31xxRegisterPortCallback( ) – register the port call back for a PHYRdyChg **871**
sil31xxSectorRW( ) – read a single sector **872**
sil31xxXbdCreate( ) – Create an XBD for the specified port. **872**
sil31xxXbdDelete( ) – Delete an XBD for a specified port **873**
sincos( ) – compute both a sine and cosine **873**
sincosf( ) – compute both a sine and cosine **874**
sinf( ) – compute a sine (ANSI) **874**
sinhf( ) – compute a hyperbolic sine (ANSI) **875**
sleep( ) – delay for a specified amount of time **875**
smMemAddToPool( ) – add memory to shared memory system partition (VxMP Option) **876**
smMemCalloc( ) – allocate memory for array from shared memory system partition (VxMP Option) **876**
smMemFindMax( ) – find largest free block in shared memory system partition (VxMP Option) **877**
smMemFree( ) – free a shared memory system partition block of memory (VxMP Option) **877**
smMemMalloc( ) – allocate block of memory from shared memory system partition (VxMP Option) **878**
smMemOptionsSet( ) – set debug options for shared memory system partition (VxMP Option) **879**
smMemRealloc( ) – reallocate block of memory from shared memory system partition (VxMP Option) **879**
smMemShow( ) – show the shared memory system partition blocks and statistics (VxMP Option) **880**
smNameAdd( ) – add a name to the shared memory name database (VxMP Option) **881**
smNameFind( ) – look up a shared memory object by name (VxMP Option) **882**
smNameFindByValue( ) – look up a shared memory object by value (VxMP Option) **883**
smNameRemove( ) – remove an object from the shared memory objects name database (VxMP Option) **884**
smNameShow( ) – show the contents of the shared memory objects name database (VxMP Option) **884**
smObjAttach( ) – attach the calling CPU to the shared memory objects facility (VxMP Option) **885**

smObjGlobalToLocal( ) – convert a global address to a local address (VxMP Option) **886**
smObjInit( ) – initialize a shared memory objects descriptor (VxMP Option) **887**
smObjLibInit( ) – install the shared memory objects facility (VxMP Option) **888**
smObjLocalToGlobal( ) – convert a local address to a global address (VxMP Option) **889**
smObjSetup( ) – initialize the shared memory objects facility (VxMP Option) **889**
smObjShow( ) – display the current status of shared memory objects (VxMP Option) **890**
smObjTimeoutLogEnable( ) – control logging of failed attempts to take a spin-lock (VxMP Option) **891**
smeRegister( ) – register with the VxBus subsystem **892**
smpLockDemo( ) – smpLockDemo entry point (shell command) **892**
snprintf( ) – write a formatted string to a buffer, not exceeding buffer size (ANSI) **893**
snsShow( ) – show information about services in the SNS directory **893**
so( ) – single-step, but step over a subroutine **895**
sp( ) – spawn a task with default parameters **895**
spinLockIsrGive( ) – release an ISR-callable spinlock **896**
spinLockIsrHeld( ) – is an ISR-callable spinlock held by the current CPU? **897**
spinLockIsrInit( ) – initialize an ISR-callable spinlock **898**
spinLockIsrTake( ) – take an ISR-callable spinlock **898**
spinLockTaskInit( ) – initialize a task-only spinlock **901**
spinLockTaskTake( ) – take a task-only spinlock **902**
sprintf( ) – write a formatted string to a buffer (ANSI) **903**
spy( ) – begin periodic task activity reports **903**
spyClkStart( ) – start collecting task activity data **904**
spyClkStop( ) – stop collecting task activity data **904**
spyHelp( ) – display task monitoring help menu **905**
spyLibInit( ) – initialize task cpu utilization tool package **905**
spyReport( ) – display task activity data **906**
spyStop( ) – stop spying and reporting **906**
spyTask( ) – run periodic task activity reports **906**
sqrtf( ) – compute a non-negative square root (ANSI) **907**
sr( ) – return the contents of the status register (SH) **907**
sr( ) – return the contents of control register **sr** (also **gbr**, **vbr**) (SH) **908**
sscanf( ) – read and convert characters from an ASCII string (ANSI) **908**
ssiDbInit( ) – Initialize SSI database. **912**
ssiShow( ) – Display SSI information **913**
ssmCompInfoGet( ) – Get component information. **913**
ssmCompRegister( ) – Register a component with SSI Manager. **914**
startupScriptFieldSplit( ) – Split the startup script field of the bootline **917**
stat( ) – get file status information using a pathname (POSIX) **917**
statfs( ) – get file status information using a pathname (POSIX) **918**
strFree( ) – free shell strings **919**
swab( ) – swap bytes **919**
symAdd( ) – create and add a symbol to a symbol table, including a group number **920**
symByValueAndTypeFind( ) – look up a symbol by value and type **921**
symByValueFind( ) – look up a symbol by value **922**
symEach( ) – call a routine to examine each entry in a symbol table **922**

symFindByName( ) – look up a symbol by name **923**
symFindByNameAndType( ) – look up a symbol by name and type **924**
symFindByValue( ) – look up a symbol by value **925**
symFindByValueAndType( ) – look up a symbol by value and type **926**
symLibInit( ) – initialize the symbol table library **927**
symRemove( ) – remove a symbol from a symbol table **927**
symShow( ) – show the symbols of specified symbol table with matching substring **928**
symShowInit( ) – initialize symbol table show routine **929**
symTblCreate( ) – create a symbol table **929**
symTblDelete( ) – delete a symbol table **930**
sysAuxClkConnect( ) – connect a routine to the auxiliary clock interrupt **930**
sysAuxClkDisable( ) – turn off auxiliary clock interrupts **931**
sysAuxClkEnable( ) – turn on auxiliary clock interrupts **931**
sysAuxClkRateGet( ) – get the auxiliary clock rate **932**
sysAuxClkRateSet( ) – set the auxiliary clock rate **932**
sysBspRev( ) – return the BSP version and revision number **933**
sysBusIntAck( ) – acknowledge a bus interrupt **933**
sysBusIntGen( ) – generate a bus interrupt **934**
sysBusTas( ) – test and set a location across the bus **935**
sysBusToLocalAdrs( ) – convert a bus address to a local address **935**
sysClkConnect( ) – connect a routine to the system clock interrupt **936**
sysClkDisable( ) – turn off system clock interrupts **936**
sysClkEnable( ) – turn on system clock interrupts **937**
sysClkRateGet( ) – get the system clock rate **937**
sysClkRateSet( ) – set the system clock rate **938**
sysHwInit( ) – initialize the system hardware **939**
sysIntDisable( ) – disable a bus interrupt level **939**
sysIntEnable( ) – enable a bus interrupt level **940**
sysLocalToBusAdrs( ) – convert a local address to a bus address **940**
sysMailboxConnect( ) – connect a routine to the mailbox interrupt **941**
sysMailboxEnable( ) – enable the mailbox interrupt **941**
sysMemTop( ) – get the address of the top of logical memory **942**
sysModel( ) – return the model name of the CPU board **942**
sysNanoDelay( ) – delay for specified number of nanoseconds **943**
sysNetMacNVRamAddrGet( ) – get network MAC address from NVRAM **943**
sysNvRamGet( ) – get the contents of non-volatile RAM **944**
sysNvRamSet( ) – write to non-volatile RAM **945**
sysPhysMemTop( ) – get the address of the top of memory **946**
sysProcNumGet( ) – get the processor number **946**
sysProcNumSet( ) – set the processor number **947**
sysScsiBusReset( ) – assert the RST line on the SCSI bus (Western Digital WD33C93 only) **947**
sysScsiConfig( ) – system SCSI configuration **948**
sysScsiInit( ) – initialize an on-board SCSI port **949**
sysSerialChanGet( ) – get the **SIO_CHAN** device associated with a serial channel **950**
sysSerialHwInit( ) – initialize the BSP serial devices to a quiesent state **950**

sysSerialHwInit2( ) – connect BSP serial device interrupts **951**
sysSerialReset( ) – reset all SIO devices to a quiet state **951**
sysToMonitor( ) – transfer control to the ROM monitor **952**
syscallDispatch( ) – dispatch a system call request to its system call handler **952**
syscallEntryHookAdd( ) – add a routine to be called on each system call entry **953**
syscallEntryHookDelete( ) – delete a previously added entry hook **954**
syscallExitHookAdd( ) – add a routine to be called on each system call exit **954**
syscallExitHookDelete( ) – delete a previously added exit hook **955**
syscallGroupRegister( ) – register a system call group with the SCI **956**
syscallHookShow( ) – display all installed system call infrastructure hooks **957**
syscallMonitor( ) – monitor system call activity **957**
syscallRegisterHookAdd( ) – add hook for system call group registration requests **958**
syscallRegisterHookDelete( ) – delete a previously added registration hook. **959**
syscallShow( ) – show registered System Call Groups, or a specific group **959**
sysctl( ) – get or set the the values of objects in the sysctl tree **961**
sysctl_add_oid( ) – add a parameter into the sysctl tree during run-time **963**
sysctl_remove_oid( ) – remove dynamically created sysctl trees **967**
sysctlbyname( ) – get or set the values of objects in the sysctl tree by name **968**
sysctlnametomib( ) – return the numeric representation of sysctl object **969**
tanf( ) – compute a tangent (ANSI) **970**
tanhf( ) – compute a hyperbolic tangent (ANSI) **970**
tarArchive( ) – archive named file/dir onto tape in tar format **971**
tarExtract( ) – extract all files from a tar formatted tape **972**
tarToc( ) – display all contents of a tar formatted tape **973**
taskActivate( ) – activate a task that has been initialized **973**
taskClose( ) – close a task **974**
taskCpuAffinityGet( ) – get the CPU affinity of a task **975**
taskCpuAffinitySet( ) – set the CPU affinity of a task **976**
taskCpuLock( ) – disable local CPU task rescheduling **978**
taskCpuUnlock( ) – enable local CPU task rescheduling **979**
taskCreate( ) – allocate and initialize a task without activation **979**
taskCreateHookAdd( ) – add a routine to be called at every task create **980**
taskCreateHookDelete( ) – delete a previously added task create routine **981**
taskCreateHookShow( ) – show the list of task create routines **981**
taskDelay( ) – delay a task from executing **982**
taskDelete( ) – delete a task **983**
taskDeleteForce( ) – delete a task without restriction **983**
taskDeleteHookAdd( ) – add a routine to be called at every task delete **984**
taskDeleteHookDelete( ) – delete a previously added task delete routine **985**
taskDeleteHookShow( ) – show the list of task delete routines **985**
taskExit( ) – exit a task **986**
taskHookShowInit( ) – initialize the task hook show facility **986**
taskIdDefault( ) – set the default task ID **987**
taskIdListGet( ) – get a list of active task IDs **987**
taskIdSelf( ) – get the task ID of a running task **988**

taskIdVerify( ) – verify the existence of a task **989**
taskInfoGet( ) – get information about a task **989**
taskInit( ) – initialize a task with a stack at a specified address **990**
taskInitExcStk( ) – initialize a task with stacks at specified addresses **992**
taskIsPended( ) – check if a task is pended **994**
taskIsReady( ) – check if a task is ready to run **995**
taskIsStopped( ) – check if a task is stopped by the debugger **995**
taskIsSuspended( ) – check if a task is suspended **996**
taskKill( ) – send a signal to a task **996**
taskLock( ) – disable task rescheduling **997**
taskName( ) – get the name associated with a task ID **998**
taskNameToId( ) – look up the task ID associated with a task name **998**
taskOpen( ) – open a task **999**
taskOpenInit( ) – initialize the task open facility **1002**
taskOptionsGet( ) – examine task options **1003**
taskOptionsSet( ) – change task options **1003**
taskPriNormalGet( ) – get the normal priority of the task **1004**
taskPriorityGet( ) – examine the priority of a task **1005**
taskPrioritySet( ) – change the priority of a task **1005**
taskRaise( ) – send a signal to the caller's task **1006**
taskRegsGet( ) – get a task's registers from the TCB **1006**
taskRegsSet( ) – set a task's registers **1007**
taskRegsShow( ) – display the contents of a task's registers **1008**
taskRestart( ) – restart a task **1008**
taskResume( ) – resume a task **1009**
taskRotate( ) – rotate ready queue for a given task priority **1010**
taskSRInit( ) – initialize the default task status register (MIPS) **1011**
taskSRSet( ) – set the task status register (MC680x0, MIPS, x86) **1011**
taskSafe( ) – make the calling task safe from deletion **1012**
taskShow( ) – display task information from TCBs **1012**
taskShowInit( ) – initialize the task show routine facility **1016**
taskSigqueue( ) – send a queued signal to a task **1016**
taskSpareFieldGet( ) – get the spare field of a TCB **1017**
taskSpareFieldSet( ) – set the spare field of a TCB **1017**
taskSpareNumAllot( ) – Allocate the first available spare field in the TCB **1018**
taskSpawn( ) – spawn a task **1019**
taskStackAllot( ) – allot memory from a task's exception stack **1021**
taskStatusString( ) – get a task's status as a string **1022**
taskSuspend( ) – suspend a task **1023**
taskSwitchHookAdd( ) – add a routine to be called at every task switch **1024**
taskSwitchHookDelete( ) – delete a previously added task switch routine **1025**
taskSwitchHookShow( ) – show the list of task switch routines **1025**
taskTcb( ) – get the task control block for a task ID **1025**
taskUnlink( ) – unlink a task **1026**
taskUnlock( ) – enable task rescheduling **1027**

taskUnsafe( ) – make the calling task unsafe from deletion **1027**
taskVarAdd( ) – add a task variable to a task **1028**
taskVarDelete( ) – remove a task variable from a task **1029**
taskVarGet( ) – get the value of a task variable **1030**
taskVarInfo( ) – get a list of task variables of a task **1031**
taskVarInit( ) – initialize the task variables facility **1031**
taskVarSet( ) – set the value of a task variable **1032**
td( ) – delete a task **1033**
tffsDevCreate( ) – create a TrueFFS block device suitable for use with dosFs **1033**
tffsDevFormat( ) – format a flash device for use with TrueFFS **1034**
tffsDevOptionsSet( ) – set TrueFFS volume options **1034**
tffsDrv( ) – initialize the TrueFFS system **1035**
tffsDrvOptionsSet( ) – set TrueFFS volume options **1035**
tffsRawio( ) – low level I/O access to flash components **1036**
ti( ) – print complete information from a task's TCB **1038**
tick64Get( ) – get the value of the kernel's tick counter as a 64 bit value **1039**
tick64Set( ) – set the value of the kernel's tick counter in 64 bits **1039**
tickAnnounce( ) – announce a clock tick to the kernel **1040**
tickAnnounceHookAdd( ) – add a hook routine to be called at each tick interrupt **1040**
tickGet( ) – get the value of the kernel's tick counter **1041**
tickSet( ) – set the value of the kernel's tick counter **1042**
timerOpenInit( ) – initialize the timer open facility **1042**
timerShowInit( ) – initialize the timer show routine facility **1043**
timer_cancel( ) – cancel a timer **1043**
timer_close( ) – close a named timer **1044**
timer_connect( ) – connect a user routine to the timer signal **1044**
timer_create( ) – allocate a timer using the specified clock for a timing base (POSIX) **1045**
timer_delete( ) – remove a previously created timer (POSIX) **1046**
timer_getoverrun( ) – return the timer expiration overrun (POSIX) **1046**
timer_gettime( ) – get the remaining time before expiration and the reload value (POSIX) **1047**
timer_modify( ) – modify a timer **1047**
timer_open( ) – open a timer **1048**
timer_settime( ) – set the time until the next expiration and arm timer (POSIX) **1050**
timer_show( ) – show information on a specified timer **1051**
timer_unlink( ) – unlink a named timer **1052**
timex( ) – time a single execution of a function or functions **1052**
timexClear( ) – clear the list of function calls to be timed **1053**
timexFunc( ) – specify functions to be timed **1054**
timexHelp( ) – display synopsis of execution timer facilities **1054**
timexInit( ) – include the execution timer library **1055**
timexN( ) – time repeated executions of a function or group of functions **1055**
timexPost( ) – specify functions to be called after timing **1056**
timexPre( ) – specify functions to be called prior to timing **1057**
timexShow( ) – display the list of function calls to be timed **1057**
tlsTaskInit( ) – Thread Local Storage init routine **1058**

tr( ) – resume a task **1058**
traceTmrResolutionGet( ) – get resolution of timestamp source, in nanoseconds **1059**
transCommit( ) – externally-callable function to do a commit **1059**
transDevCreate( ) – create a transactional XBD. **1060**
trgAdd( ) – add a new trigger to the trigger list **1060**
trgChainSet( ) – chains two triggers **1062**
trgDelete( ) – delete a trigger from the trigger list **1062**
trgDisable( ) – turn a trigger off **1063**
trgEnable( ) – enable a trigger **1063**
trgEvent( ) – trigger a user-defined event **1064**
trgLibInit( ) – initialize the triggering library **1064**
trgOff( ) – set triggering off **1065**
trgOn( ) – set triggering on **1065**
trgReset( ) – Reset a trigger in the trigger list **1066**
trgShow( ) – show trigger information **1066**
trgShowInit( ) – initialize the trigger show facility **1067**
trgWorkQReset( ) – Resets the trigger work queue task and queue **1067**
trunc( ) – truncate to integer **1068**
truncf( ) – truncate to integer **1068**
ts( ) – suspend a task **1069**
tsecRegister( ) – register with the VxBus subsystem **1069**
tt( ) – display a stack trace of a task **1070**
ttyDevCreate( ) – create a VxWorks device for a serial channel **1071**
ttyDrv( ) – initialize the *tty* driver **1071**
tw( ) – print info about the object the given task is pending on **1072**
tyAbortFuncSet( ) – set the abort function **1072**
tyAbortGet( ) – get the abort character **1073**
tyAbortSet( ) – change the abort character **1073**
tyBackspaceSet( ) – change the backspace character **1074**
tyDeleteLineSet( ) – change the line-delete character **1074**
tyDevInit( ) – initialize the *tty* device descriptor **1075**
tyDevRemove( ) – remove the *tty* device descriptor **1076**
tyDevTerminate( ) – terminate the *tty* device descriptor **1076**
tyEOFGet( ) – get the current end-of-file character **1077**
tyEOFSet( ) – change the end-of-file character **1077**
tyIRd( ) – interrupt-level input **1078**
tyITx( ) – interrupt-level output **1078**
tyIoctl( ) – handle device control requests **1079**
tyLibInit( ) – initialize the *tty* library **1079**
tyMonitorTrapSet( ) – change the trap-to-monitor character **1080**
tyRead( ) – do a task-level read for a *tty* device **1081**
tyWrite( ) – do a task-level write for a *tty* device **1081**
tyXoffHookSet( ) – install a hardware flow control function **1082**
unixDiskDevCreate( ) – create a UNIX disk device **1082**
unixDiskInit( ) – initialize a dosFs disk on top of UNIX **1083**

unixDrv( ) – install UNIX disk driver **1084**
unld( ) – unload an object module by specifying a file name or module ID (shell command) **1084**
unldByGroup( ) – unload an object module by specifying a group number **1085**
unldByModuleId( ) – unload an object module by specifying a module ID **1086**
unldByNameAndPath( ) – unload an object module by specifying a name and path **1087**
unlink( ) – unlink a file **1088**
unstatShow( ) – display all **AF_LOCAL** sockets **1088**
usrClock( ) – user-defined system clock interrupt routine **1089**
usrFdiskPartCreate( ) – create an FDISK-like partition table on a disk **1090**
usrFdiskPartRead( ) – read an FDISK-style partition table **1091**
usrFdiskPartShow( ) – parse and display partition data **1092**
usrFormatTrans( ) – Perform a low-level trans XBD format operation **1092**
usrIdeConfig( ) – mount a DOS file system from an IDE hard disk **1093**
usrInit( ) – user-defined system initialization routine **1094**
usrRoot( ) – the root task **1094**
usrScsiConfig( ) – configure SCSI peripherals **1095**
usrTransCommit( ) – Set a transaction point on a trans XBD **1096**
usrTransCommitFd( ) – set a transaction point using a file descriptor **1096**
uswab( ) – swap bytes with buffers that are not necessarily aligned **1097**
utf16ToCP( ) – Convert a UTF-16 encoded Unicode character to a codepoint. **1097**
utf16ToUtf8String( ) – Convert a UTF-16 string to a UTF-8 String **1098**
utf16ToUtf8StringBOM( ) – Convert UTF-16 to UTF-8 based on a Byte Order Mark **1098**
utf8ToCP( ) – Convert a UTF-8 encoded Unicode character to the Unicode codepoint. **1099**
utf8ToUtf16String( ) – convert a UTF-8 string to a UTF-16 string **1099**
utf8ToUtf16StringBOM( ) – Convert UTF-8 to UTF16 with a Byte Order Mark **1100**
utfLibInit( ) – initialize the UTF library **1101**
utflen16( ) – Return the number of 16-bit words used by a UTF-16 encoding. **1101**
utflen8( ) – return the encoding length of a **NULL** terminated UTF-8 string **1101**
utime( ) – update time on a file **1102**
valloc( ) – allocate memory on a page boundary from the kernel heap **1102**
version( ) – print VxWorks version information **1103**
vfdprintf( ) – write a string formatted with a variable argument list to a file descriptor **1104**
virtualDiskClose( ) – close a virtual disk block device. **1104**
virtualDiskCreate( ) – create a virtual disk device. **1105**
virtualDiskInit( ) – install the virtual disk driver **1105**
vmArch32LibInit( ) – initialize the arch specific unbundled VM library (VxVMI Option) **1106**
vmArch32Map( ) – map 32bit physical space into 32bit virtual space (VxVMI Option) **1106**
vmArch32Translate( ) – translate a 32bit virtual address to a 32bit physical address (VxVMI Option) **1107**
vmArch36LibInit( ) – initialize the arch specific unbundled VM library (VxVMI Option) **1108**
vmArch36Map( ) – map 36bit physical space into 32bit virtual space (VxVMI Option) **1108**
vmArch36Translate( ) – translate a 32bit virtual address to a 36bit physical address (VxVMI Option) **1109**
vmAttrShow( ) – display the text representation of a MMU attribute value **1110**
vmBaseArch32LibInit( ) – initialize the arch specific bundled VM library **1110**
vmBaseArch32Map( ) – map 32bit physical to the 32bit virtual memory **1111**
vmBaseArch32Translate( ) – translate a 32bit virtual address to a 32bit physical address **1111**

vmBaseArch36LibInit( ) – initialize the arch specific bundled VM library **1112**
vmBaseArch36Map( ) – map 36bit physical to the 32bit virtual memory **1112**
vmBaseArch36Translate( ) – translate a 32bit virtual address to a 36bit physical address **1113**
vmBaseGlobalMapInit( ) – initialize global mapping (obsolete) **1113**
vmBasePageSizeGet( ) – return the MMU page size (obsolete) **1114**
vmBaseStateSet( ) – change the state of a block of virtual memory (obsolete) **1114**
vmContextShow( ) – display the translation table for a context **1115**
vmGlobalMapInit( ) – initialize global mapping **1116**
vmMap( ) – map physical space into virtual space **1117**
vmPageLock( ) – lock the pages. **1118**
vmPageMap( ) – map physical space into virtual space **1119**
vmPageOptimize( ) – Optimize the address range if possible. **1120**
vmPageSizeGet( ) – return the page size **1121**
vmPageUnlock( ) – unlock the pages. **1121**
vmPhysTranslate( ) – translate a physical address to a virtual address **1122**
vmStateGet( ) – get the state of a page of virtual memory **1122**
vmStateSet( ) – change the state of a block of virtual memory **1123**
vmTextProtect( ) – write-protect kernel text segment **1125**
vmTranslate( ) – translate a virtual address to a physical address **1125**
voprintf( ) – write a formatted string to an output function **1126**
vprintf( ) – write a string formatted with a variable argument list to standard output (ANSI) **1127**
vrfsDevCreate( ) – Instantiate the VRFS **1127**
vrfsInit( ) – Initialize the Virtual Root File System Library **1128**
vsnprintf( ) – write a string formatted with a variable argument list to a buffer, not exceeding buffer size (ANSI) **1128**
vsprintf( ) – write a string formatted with a variable argument list to a buffer (ANSI) **1129**
vxAtomicAdd( ) – atomically add a value to a memory location **1129**
vxAtomicAnd( ) – atomically perform a bitwise AND on a memory location **1130**
vxAtomicClear( ) – atomically clear a memory location **1130**
vxAtomicDec( ) – atomically decrement a memory location **1131**
vxAtomicGet( ) – atomically get a memory location **1132**
vxAtomicInc( ) – atomically increment a memory location **1132**
vxAtomicNand( ) – atomically perform a bitwise NAND on a memory location **1133**
vxAtomicOr( ) – atomically perform a bitwise OR on memory location **1133**
vxAtomicSet( ) – atomically set a memory location **1134**
vxAtomicSub( ) – atomically subtract a value from a memory location **1134**
vxAtomicXor( ) – atomically perform a bitwise XOR on a memory location **1135**
vxCas( ) – atomically compare-and-swap the contents of a memory location **1135**
vxCpuConfiguredGet( ) – get the number of configured CPUs in the system **1136**
vxCpuEnabledGet( ) – get a set of running CPUs **1136**
vxCpuIndexGet( ) – get the index of the calling CPU **1138**
vxCr0Get( ) – get a content of the Control Register 0 (x86) **1138**
vxCr0Set( ) – set a value to the Control Register 0 (x86) **1139**
vxCr2Get( ) – get a content of the Control Register 2 (x86) **1139**
vxCr2Set( ) – set a value to the Control Register 2 (x86) **1140**

vxCr3Get( ) – get a content of the Control Register 3 (x86) **1140**
vxCr3Set( ) – set a value to the Control Register 3 (x86) **1140**
vxCr4Get( ) – get a content of the Control Register 4 (x86) **1141**
vxCr4Set( ) – set a value to the Control Register 4 (x86) **1141**
vxDrGet( ) – get a content of the Debug Register 0 to 7 (x86) **1142**
vxDrSet( ) – set a value to the Debug Register 0 to 7 (x86) **1142**
vxEflagsGet( ) – get a content of the EFLAGS register (x86) **1143**
vxEflagsSet( ) – set a value to the EFLAGS register (x86) **1143**
vxGdtrGet( ) – get a content of the Global Descriptor Table Register (x86) **1144**
vxIdtrGet( ) – get a content of the Interrupt Descriptor Table Register (x86) **1144**
vxLdtrGet( ) – get a content of the Local Descriptor Table Register (x86) **1144**
vxMemArchProbe( ) – architecture specific part of vxMemProbe **1145**
vxMemProbe( ) – probe an address for a bus error **1145**
vxMemProbe( ) – probe an address for a bus error **1146**
vxMemProbeInit( ) – add vxMemProbeTrap exception handler to exc handler chain **1148**
vxPowerDown( ) – place the processor in reduced-power mode (PowerPC, SH) **1148**
vxPowerModeGet( ) – get the power management mode (PowerPC, SH, x86) **1148**
vxPowerModeSet( ) – set the power management mode (PowerPC, SH, x86) **1149**
vxSSDisable( ) – disable the superscalar dispatch (MC68060) **1150**
vxSSEnable( ) – enable the superscalar dispatch (MC68060) **1151**
vxTas( ) – C-callable atomic test-and-set primitive **1151**
vxTssGet( ) – get a content of the TASK register (x86) **1152**
vxTssSet( ) – set a value to the TASK register (x86) **1152**
vxbFileNvRamGet( ) – get the contents of non-volatile RAM **1153**
vxbFileNvRamRegister( ) – register vxbFileNvRam driver **1154**
vxbFileNvRamSet( ) – write to non-volatile RAM **1154**
vxbFileNvRampDrvCtrlShow( ) – show pDrvCtrl for template controller **1155**
vxbIntelIchStorageRegister( ) – register driver with vxbus **1155**
vxbNonVolGet( ) – get the contents of non-volatile RAM **1156**
vxbNonVolLibInit( ) – Non Volatile RAM library initialization **1156**
vxbNonVolSet( ) – write to non-volatile memory **1157**
vxbSI31xxStorageRegister( ) – register driver with vxbus **1157**
vxsimHostCpuVarsInit( ) – intialize per cpu variable pointers **1158**
vxsimHostDllLoad( ) – load the given Dll to VxSim. **1158**
vxsimHostMmuCurrentSet( ) – set current translation table mapping **1159**
vxsimHostMmuProtect( ) – set/clear protection on mmu pages **1159**
vxsimHostProcAddrGet( ) – return the address of a host API **1160**
vxsimHostProcCall( ) – call a host routine **1160**
vxsimHostSioBaudRateSet( ) – set SIO device transfert rate **1161**
vxsimHostSioClose( ) – close SIO device **1161**
vxsimHostSioIntVecGet( ) – get SIO device interrupt vector **1162**
vxsimHostSioModeSet( ) – set SIO device mode (poll/interrupt) **1162**
vxsimHostSioOpen( ) – open SIO device **1163**
vxsimHostSioRead( ) – read SIO device into buffer **1163**
vxsimHostSioWrite( ) – write buffer to SIO device **1164**

w( ) – print a summary of each task's pending information, task by task **1164**
wdCancel( ) – cancel a currently counting watchdog **1165**
wdCreate( ) – create a watchdog timer **1165**
wdDelete( ) – delete a watchdog timer **1166**
wdInitialize( ) – initialize a pre-allocated watchdog. **1166**
wdShow( ) – show information about a watchdog **1167**
wdShowInit( ) – initialize the watchdog show facility **1168**
wdStart( ) – start a watchdog timer **1168**
wdbMdlSymSyncLibInit( ) – initialize modules and symbols synchronization library **1169**
wdbSystemSuspend( ) – suspend the system **1169**
wdbUserEvtLibInit( ) – include the WDB user event library **1170**
wdbUserEvtPost( ) – post a user event string to host tools **1171**
wim( ) – return the contents of the window invalid mask register (SimSolaris) **1172**
windPwrDownRtnSet( ) – register a BSP power-down function **1172**
windPwrModeGet( ) – Get the current power mode **1173**
windPwrModeSet( ) – Set the BSP power mode **1174**
windPwrUpRtnSet( ) – register a BSP power-up function **1174**
write( ) – write bytes to a file **1175**
wvAllObjsSet( ) – set instrumented state for all objects and classes **1176**
wvCurrentLogGet( ) – return a pointer to the currently active System Viewer log **1177**
wvCurrentLogListGet( ) – return a pointer to the System Viewer log list **1177**
wvCurrentLogListSet( ) – set the current log list **1177**
wvCurrentLogSet( ) – select a System Viewer log as currently active **1178**
wvEdrInst( ) – instrument ED&R Events **1178**
wvEvent( ) – log a user-defined event **1179**
wvEventInst( ) – instrument VxWorks Events **1179**
wvEvtClassClear( ) – clear the specified class of events from those being logged **1180**
wvEvtClassClearAll( ) – clear all classes of events from those logged **1181**
wvEvtClassGet( ) – get the current set of classes being logged **1181**
wvEvtClassSet( ) – set the class of events to log **1181**
wvEvtLogStart( ) – start logging events to the buffer **1182**
wvEvtLogStop( ) – stop logging events to the buffer **1182**
wvFileUploadPathCreate( ) – create a file for depositing event data **1183**
wvFileUploadPathLibInit( ) – initialize the **wvFileUploadPathLib** library **1183**
wvFileUploadPathWrite( ) – write to the event-destination file **1184**
wvLibInit( ) – initialize **wvLib** - first step **1184**
wvLibInit2( ) – initialize **wvLib** - final step **1184**
wvLogCountGet( ) – return the number of logs in the curent log list **1185**
wvLogCreate( ) – Create a System Viewer log **1185**
wvLogDelete( ) – Delete a System Viewer log **1186**
wvLogFirstGet( ) – return a pointer to the first log in the System Viewer log list **1186**
wvLogListCreate( ) – create a list to hold System Viewer logs **1187**
wvLogListDelete( ) – delete a System Viewer log list **1187**
wvLogNextGet( ) – return a pointer to the next log in the System Viewer log list **1188**
wvObjInst( ) – instrument objects **1188**

wvObjInstModeSet( ) – set object instrumentation on/off **1189**
wvPartitionGet( ) – determine partition in use for System Viewer logging **1190**
wvPartitionSet( ) – specify a partition for use by System Viewer logging **1190**
wvRBuffMgrPrioritySet( ) – set the priority of the System Viewer rBuff manager **1191**
wvSalInst( ) – instrument SAL **1191**
wvSigInst( ) – instrument signals **1192**
wvSockUploadPathClose( ) – close the socket upload path **1193**
wvSockUploadPathCreate( ) – establish an upload path to the host using a socket **1193**
wvSockUploadPathLibInit( ) – initialize **wvSockUploadPathLib** library **1194**
wvSockUploadPathWrite( ) – write to the socket upload path **1194**
wvTmrRegister( ) – register a timestamp timer **1195**
wvTsfsUploadPathClose( ) – close the TSFS-socket upload path **1196**
wvTsfsUploadPathCreate( ) – open an upload path to the host using a TSFS socket **1196**
wvTsfsUploadPathLibInit( ) – initialize **wvTsfsUploadPathLib** library **1197**
wvTsfsUploadPathWrite( ) – write to the TSFS upload path **1197**
wvUploadStart( ) – start upload of events to the host **1198**
wvUploadStop( ) – stop upload of events to host **1198**
wvUploadTaskConfig( ) – set priority and stacksize of **tWVUpload** task **1199**
xattrib( ) – modify MS-DOS file attributes of many files **1199**
xbdBlkDevCreate( ) – create an XBD block device wrapper **1200**
xbdBlkDevCreateSync( ) – synchronously create an XBD block device wrapper **1201**
xbdBlkDevDelete( ) – deletes an XBD block device wrapper **1201**
xbdBlkDevLibInit( ) – initialize the XBD block device wrapper **1202**
xbdCbioDevCreate( ) – create an XBD CBIO device wrapper **1202**
xbdCbioDevDelete( ) – deletes an XBD CBIO device wrapper **1203**
xbdCbioLibInit( ) – initialize the XBD block device wrapper **1203**
xbdCreatePartition( ) – partition an XBD device **1204**
xbdRamDiskDevCreate( ) – create an XBD ram disk **1205**
xbdRamDiskDevDelete( ) – XBD Ram Disk Deletion routine **1205**
xbdTransDevCreate( ) – create a transactional XBD. **1206**
xbdTransInit( ) – initialize the transactional XBD subsystem. **1206**
xcopy( ) – copy a hierarchy of files with wildcards **1206**
xdelete( ) – delete a hierarchy of files with wildcards **1207**
y( ) – return the contents of the y register (SimSolaris) **1208**
ykRegister( ) – register with the VxBus subsystem **1208**
ynRegister( ) – register with the VxBus subsystem **1208**

CPToUtf16( )

NAME

CPToUtf16( ) – Convert a Unicode codepoint to a UTF-16 encoding

SYNOPSIS

```
int CPToUtf16
    (
    const unsigned long codePoint,
    unsigned short *    utf16,
    const int           length,       /* Length is in 16-bit words */
    const int           littleEndian
    )
```

DESCRIPTION

This routine converts an unsigned long representing the value of a Unicode codepoint to the UTF-16 encoding.

RETURNS

If positive, the return value indicates the number of words used to encode this codepoint. If non-positive, the return value of **UC_FORMAT** indicates that the value to be converted is not a legitimate Unicode codepoint. A return value of **UC_BUFFER** indicates that the output string is of insufficient length to hold the converted character.

ERRNO

Not Available

SEE ALSO

utfLib

CPToUtf8( )

NAME

CPToUtf8( ) – Convert a Unicode codepoint to a UTF-8 encoding

SYNOPSIS

```
int CPToUtf8
    (
    const unsigned long codePoint,
    unsigned char *     utf8,
    const int           length
    )
```

DESCRIPTION

This routine converts an unsigned long representing the value of a Unicode codepoint to the UTF-8 encoding.

RETURNS

If positive, the return value indicates the number of bytes used to encode this codepoint. If non-positive, the return value of **UC_FORMAT** indicates that the value to be converted is not a legitimate Unicode codepoint. A return value of **UC_BUFFER** indicates that the output string is of insufficient length to hold the converted character.

ERRNO Not Available

SEE ALSO **utfLib**

CPUSET_ATOMICCLR( )

NAME **CPUSET_ATOMICCLR( )** – atomically clear a CPU from a CPU set

SYNOPSIS

```
CPUSET_ATOMICCLR
    (
    cpuset        /* CPU set to operate on */
    n             /* index of CPU to clear */
    )
```

DESCRIPTION This macro atomically clears CPU index *n* from the *cpuset* variable. The status of other CPU indices in the set, whether set or cleared, is not a affected by this action. This action is the reverse of what **CPUSET_ATOMICSET** does. Atomic clearing of a CPU in a set is necessary when the set is likely to be manipulated by more than one task or ISR.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable and the CPU index is always an unsigned integer between 0 and the number of CPUs either enabled or configured in the system. APIs that expect a cpuset_t variable as an argument describe the restrictions that apply.

RETURNS N/A

ERRNO N/A

SEE ALSO **cpuset**, **CPUSET_CLR**, **CPUSET_ATOMICSET**

CPUSET_ATOMICCOPY( )

NAME **CPUSET_ATOMICCOPY( )** – atomically copy a CPU set value

SYNOPSIS

```
CPUSET_ATOMICCLR
    (
    cpusetDst,         /* cpuset to copy to */
    cpusetSrc             /* cpuset to copy from */
    )
```

DESCRIPTION

This macro atomically copies the bit sets from *cpusetSrc* cpuset and stores the copy in the *cpusetDst* variable.

While this macro does not enforce any restrictions, it is expected that *cpusetSrc* and *cpusetDst* are cpuset_t type variables. APIs that expect a cpuset_t variable as an argument describe the restrictions that apply.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

cpuset

CPUSET_ATOMICSET( )

NAME

CPUSET_ATOMICSET( ) – atomically set a CPU in a CPU set

SYNOPSIS

```
CPUSET_ATOMICSET
    (
    cpuset       /* CPU set to operate on */
    n            /* index of CPU to set */
    )
```

DESCRIPTION

This macro atomically sets CPU index *n* in the *cpuset* variable. It is the atomic version of **CPUSET_SET**. The status of other CPU indices in the set, whether set or cleared, is not affected by this action. For example, to set CPU0 and CPU1 in a set, this macro needs to be used twice specifying n=0 and then n=1. Atomic setting of a CPU in a set is necessary when the set is likely to be manipulated by more than one task or ISR.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable and the CPU index is always an unsigned integer between 0 and the number of CPUs either enabled or configured in the system. APIs that expect a cpuset_t variable as an argument describe the restrictions that apply.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

cpuset, **CPUSET_SET**, **CPUSET_ATOMICCLR**

CPUSET_CLR( )

NAME

CPUSET_CLR( ) – clear a CPU from a CPU set

SYNOPSIS

```
CPUSET_CLR
    (
    cpuset       /* CPU set to operate on */
    n            /* index of CPU to clear */
    )
```

DESCRIPTION

This macro clears CPU index *n* in the *cpuset* variable. The status of other CPU indices in the set, whether set or cleared, is not affected by this action. This action is the reverse of what **CPUSET_SET** does.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable and the CPU index is an unsigned integer between 0 and the number of CPUs either enabled or configured in the system. APIs that expect a cpuset_t variable as an argument describe the restrictions that apply.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

cpuset, **CPUSET_ZERO**, **CPUSET_ISZERO**, **CPUSET_SET**, **vxCpuConfiguredGet( )**

CPUSET_ISSET( )

NAME

CPUSET_ISSET( ) – determine if a CPU is set in a CPU set

SYNOPSIS

```
CPUSET_ISSET
    (
    cpuset       /* CPU set to operate on */
    n            /* index of CPU to query */
    )
```

DESCRIPTION

This macro resolves to **TRUE** if the index of CPU *n* is set in *cpuset*. Otherwise it returns **FALSE**.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable.

RETURNS

Macro resolves to **TRUE** or **FALSE**.

ERRNO N/A

SEE ALSO **cpuset**, **CPUSET_SET**, **CPUSET_SETALL**, **CPUSET_SETALL_BUT_SELF**

CPUSET_ISZERO( )

NAME **CPUSET_ISZERO( )** – determine if all CPUs are cleared from a CPU set

SYNOPSIS

```
CPUSET_ISZERO
    (
    cpuset       /* CPU set to operate on */
    )
```

DESCRIPTION This macro returns **TRUE** if variable *cpuset* is empty of CPU indices. Otherwise it returns **FALSE**.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable.

RETURNS Macro resolves to **TRUE** or **FALSE**.

ERRNO N/A

SEE ALSO **cpuset**, **CPUSET_ZERO**

CPUSET_SET( )

NAME **CPUSET_SET( )** – set a CPU in a CPU set

SYNOPSIS

```
CPUSET_SET
    (
    cpuset,       /* CPU set to operate on */
    n             /* index of CPU to set */
    )
```

DESCRIPTION This macro sets CPU index *n* in the *cpuset* variable. The status of other CPU indices in the set, whether set or cleared, is not affected by this action. For example, to set CPU0 and CPU1 in a set, this macro needs to be used twice specifying n=0 and then n=1.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable and the CPU index is an unsigned integer between 0 and the number of CPUs either enabled or configured in the system. APIs that expect a cpuset_t variable as an argument describe the restrictions that apply.

RETURNS N/A

ERRNO N/A

SEE ALSO **cpuset**, **CPUSET_SETALL**, **CPUSET_SETALL_BUT_SELF**, **CPUSET_ISSET**

CPUSET_SETALL( )

NAME **CPUSET_SETALL( )** – set all CPUs in a CPU set

SYNOPSIS

```
CPUSET_SETALL
    (
    cpuset        /* CPU set to operate on */
    )
```

DESCRIPTION This macro sets the *cpuset* variable with the index of every CPU that is configured in the system.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable.

RETURNS N/A

ERRNO N/A

SEE ALSO **cpuset**, **CPUSET_SET**, **CPUSET_SETALL_BUT_SELF**, **CPUSET_ISSET**, **vxCpuConfiguredGet( )**

CPUSET_SETALL_BUT_SELF( )

NAME **CPUSET_SETALL_BUT_SELF( )** – set all CPUs except self in CPU set

SYNOPSIS

```
CPUSET_SETALL_BUT_SELF
    (
    cpuset       /* CPU set to operate on */
    )
```

DESCRIPTION

This macro sets the *cpuset* variable with the index of every CPU that is configured in the system excluding the index of the CPU that is calling this macro. Users must be aware that after a cpuset_t variable is set using this macro, the CPU index of the calling CPU, the one that is excluded from the cpuset, may no longer be the correct one when the variable is subsequently used, because a scheduling event may have taken the CPU away from the caller of the macro. The following scenario describes how this can take place:

- Task 1, running on CPU0, sets myCpuSet using **CPUSET_SETALL_BUT_SELF**. This causes myCpuSet to be set with the index of every configured CPU except CPU0.
- An interrupt occurs and causes a scheduling event that makes Task 2 run on CPU0, effectively making Task 1 wait for a CPU to become available.
- Another scheduling event occurs freeing CPU1 allowing Task 1 to resume execution. At this point the excluded index in myCpuSet is that of CPU0, not that of CPU1, which is the CPU on which task1 now runs.

Conceptually this is the same issue as explained in the reference entry for **vxCpuIndexGet( )**. One solution to this problem is for a task to use **taskCpuLock( )** to prevent it from migrating to another CPU while is makes use of a cpuset that has been set with **CPUSET_SETALL_BUT_SELF**. This issue does not exist for ISRs since these never migrate from one CPU to another while they are executing.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

cpuset, **CPUSET_SET**, **CPUSET_SETALL**, **CPUSET_ISSET**, **vxCpuIndexGet( )**, **vxCpuConfiguredGet( )**

CPUSET_ZERO( )

NAME

CPUSET_ZERO( ) – clear all CPUs from a CPU set

SYNOPSIS

```
CPUSET_ZERO
    (
    cpuset       /* CPU set to operate on */
    )
```

DESCRIPTION

This macro clears all CPU indices from the *cpuset* variable. This action is the reverse of what **CPUSET_SETALL** does.

While this macro does not enforce any restrictions, it is expected that cpuset is always a cpuset_t type variable.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
cpuset, **CPUSET_CLR**, **CPUSET_ISZERO**, **CPUSET_SETALL**

Sysctl( )

NAME
Sysctl( ) – get or set values for kernel state variables from the C shell

SYNOPSIS
```
STATUS Sysctl
    (
    char *cmd
    )
```

DESCRIPTION
Sysctl is a C shell utility that can be used to get or set the values of various run-time parameters in the system. The underlying calls for this utility are the same as for sysctl and sysctlbyname. Sysctl can be used to get or set simple INTEGER type variables. It does not have the ability to retrive complex data such as structures and executing procedures. Structures such as "icmp stats" are displayed as hex dumps. To configure these complex variables, one must use either "sysctl" or "sysctlbyname".

USAGE
To get the value of a variable, use
    Sysctl ("net.inet.ip.forwarding")

To set the value of a variable, use
    Sysctl ("net.inet.ip.forwarding=0")

To get a listing of all the variables registered in the system,
    Sysctl ("-A")

Options: -A   Equivalent to -o -a (for compatibility).

-a   List all the currently available non-opaque values. This option
      is ignored if one or more variable names are specified on the
      command line.

-b Force the value of the variable(s) to be output in raw, binary format. No names are printed and no terminating newlines are output. This is mostly useful with a single variable.

-e Separate the name and the value of the variable(s) with =. This is useful for producing output which can be fed back to the sysctl utility. This option is ignored if either -N or -n is specified, or a variable is being set.

-N Show only variable names, not their values.

-n Show only variable values, not their names.

-o Show opaque variables (which are normally suppressed). The format and length are printed, as well as a hex dump of the first sixteen bytes of the value.

-X Equivalent to -x -a (for compatibility).

-x As -o, but prints a hex dump of the entire value instead of just the first few bytes.

RETURNS
OK upon success or **ERROR** if an error occurred

ERRNO
N/A

SEE ALSO
sysctl

VX_MEM_BARRIER_R( )

NAME
VX_MEM_BARRIER_R( ) – Read Memory Barrier

SYNOPSIS
```
void VX_MEM_BARRIER_R
    (
    void
    )
```

DESCRIPTION
This routine is the read memory barrier that guarantees all load memory operations before the read memory barrier has occurred before any subsequent load operations after the read barrier.

A read barrier does not necessarily have the same effect for store operations; hence a read memory barrier should not be used to ensure ordering of store operations.

Generally, a read memory barrier is paired with a write memory barrier in a multicore system to ensure proper interactions between CPUs. Here is an example of read and write barrier used in a system to ensure proper reading and writing of memory.

Example:

```
CPU 1           CPU 2
aa = value1
VX_MEM_BARRIER_W()
bb = value2
                cc = aa
                VX_MEM_BARRIER_R()
                dd = bb
```

In the above example, the write of aa must be enforced before CPU 2 can load the value *value1* of aa. Otherwise, the wrong value of aa might be loaded. The read barrier is necessary to ensure that the read of aa is performed before performing the read of bb.

RETURNS N/A

ERRNOS N/A

SEE ALSO **vxAtomicLib**

VX_MEM_BARRIER_RW( )

NAME **VX_MEM_BARRIER_RW( )** – Read/Write Memory Barrier

SYNOPSIS

```
void VX_MEM_BARRIER_RW
    (
    void
    )
```

DESCRIPTION This routine provides the read/write memory barrier. A read/write memory barrier is a general barrier that enforces ordering for both reads and writes. Hence, this read/write memory barrier can be used as a substitution for both read and write barriers.

RETURNS N/A

ERRNOS N/A

SEE ALSO **vxAtomicLib**

VX_MEM_BARRIER_W( )

NAME

VX_MEM_BARRIER_W( ) – Write memory barrier

SYNOPSIS

```
void VX_MEM_BARRIER_W
    (
    void
    )
```

DESCRIPTION

This routine is the write memory barrier that guarantees all store memory operations before the write barrier has occurred before any subsequent store operations after the barrier.

A write barrier does not necessarily have the same effect for load operations; hence a write memory barrier should not be used to ensure ordering of load operations.

This routine is essential to ensure proper accesses to memory that are shared between CPUs in an SMP system since this enforces the ordering of memory accesses.

RETURNS

N/A

ERRNOS

N/A

SEE ALSO

vxAtomicLib

a0( )

NAME

a0( ) – return the contents of register **a0** (also **a1** - **a7**) (MC680x0)

SYNOPSIS

```
int a0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **a0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all address registers (**a0** - **a7**): **a0( )** - **a7( )**.

The stack pointer is accessed via **a7( )**.

RETURNS

The contents of register **a0** (or the requested register).

ERRNO

Not Available

SEE ALSO **dbgArchLib**, the VxWorks programmer guides.

access( )

NAME **access( )** – determine accessibility of a file

SYNOPSIS

```
int access
    (
    const char *path,  /* path of the file */
    int         amode   /* access mode to query */
    )
```

DESCRIPTION The **access( )** function checks the file named by the pathname pointed to by the *path* argument for accessibility according to the bit pattern contained in *amode*, This allows a process, RTP to verify that it has permission to access this file.

The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked (**R_OK**, **W_OK**, **X_OK**) or the existence test, **F_OK**.

If any access permissions are to be checked, each will be checked individually. If the process has appropriate privileges, it may indicate success even if none of the related permission bits is set.

These constants are defined in *unistd.h* as follows:

R_OK
Test for read permission.

W_OK
Test for write permission.

X_OK
Test for execute or search permission.

F_OK
Check existence of file

RETURNS If the requested access is permitted, **access( )** succeeds and returns **OK**, 0. Otherwise, **ERROR**, -1 is returned and errno is set to indicate the error.

ERRNO

ENOENT
Either *path* is an empty string or **NULL** pointer.

ELOOP
Circular symbolic link of *path*, or too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in *path*.

others
Other errors reported by device driver of *path*.

SEE ALSO **fsPxLib**

acosf( )

NAME **acosf( )** – compute an arc cosine (ANSI)

SYNOPSIS
```
float acosf
    (
    float x  /* number between -1 and 1 */
    )
```

DESCRIPTION This routine computes the arc cosine of *x* in single precision. If *x* is the cosine of an angle *T*, this function returns *T*.

RETURNS The single-precision arc cosine of *x* in the range 0 to pi radians.

ERRNO Not Available

SEE ALSO **mathALib**

adrSpaceInfoGet( )

NAME **adrSpaceInfoGet( )** – get status of the address space library

SYNOPSIS
```
STATUS adrSpaceInfoGet
    (
    ADR_SPACE_INFO * pInfo  /* address space info pointer */
    )
```

DESCRIPTION This routine initializes an **ADR_SPACE_INFO** structure parameter, *pInfo*, with the current state of the address space library.

The following information is returned:

physAllocUnit
allocation unit (page) size in physical space.

physTotalPages
total system RAM pages.

physFreePages
unmapped system RAM pages.

physMaxSize
largest unmapped system RAM block.

kernelAllocUnit
allocation unit (page) size in kernel region.

kernelTotalPages
total pages in the kernel region .

kernelFreePages
unmapped pages in the kernel region.

kernelMaxSize
largest unmapped block in kernel region.

userAllocUnit
allocation unit (page) size in user region.

userTotalPages
total pages in the user region.

userFreePages
unmapped pages in the user region.

userMaxSize
largest unmapped block in user region.

RETURNS **OK**, or **ERROR** in case of failure.

ERRNO Not Available

SEE ALSO **adrSpaceLib**, **adrSpaceShow**

adrSpacePageUnmap( )

NAME **adrSpacePageUnmap( )** – unmap a set of virtual pages

SYNOPSIS
```
STATUS adrSpacePageUnmap
```

```
    (
    VM_CONTEXT_ID vmContext,  /* VM context ID */
    VIRT_ADDR     virtAdr,    /* base virtual address */
    UINT          numPages,   /* pages to unmap */
    UINT          options     /* unmap options */
    )
```

DESCRIPTION This routine unmaps *numVirtPages* virtual pages, starting at *virtAdr*. The associated physical pages are returned to the physical page pool while the virtual pages remain allocated.

RETURNS **OK** on success, and **ERROR** otherwise.

ERRNOS The routine may set the following errnos:

S_adrSpaceLib_SIZE_IS_INVALID
The *numPages* parameter is 0.

S_adrSpaceLib_PARAMETER_NOT_ALIGNED
The parameter *virtAdr* is not aligned on a page boundary.

S_pgMgrLib_VIRT_ADDR_OUT_OF_RANGE
The specified virtual address range, starting at *virtAdr*, and of size *numVirtPages* is not in the range of the RTP's virtual region.

SEE ALSO **adrSpaceLib**, **pgMgrPageMap( )**, **pgMgrPageFree( )**.

adrSpaceRAMAddToPool( )

NAME **adrSpaceRAMAddToPool( )** – add specified memory block to RAM pool

SYNOPSIS

```
STATUS adrSpaceRAMAddToPool
    (
    PHYS_ADDR startAddr,  /* start adress of RAM memory block */
    UINT      ramSize     /* in bytes */
    )
```

DESCRIPTION This routine adds the specified RAM to the system global RAM page pool. The address range of the RAM must not partially or fully be part of the physical page pool, and must not be mapped.

The length and start address are expected to be MMU page aligned. The smallest amount of space that can be added is one page.

RETURNS **OK** on success and **ERROR** otherwise.

ERRNO Possible errnos generated by this routine include:

S_adrSpaceLib_SIZE_IS_INVALID
ramSize is less than a page.

S_adrSpaceLib_PARAMETER_NOT_ALIGNED
start or end address is not page size aligned.

S_adrSpaceLib_PHYSICAL_OVERLAP
block overlaps with memory already in the RAM pool

S_adrSpaceLib_ADDRESS_OUT_OF_RANGE
address out of 4GBytes range.

SEE ALSO **adrSpaceLib**, **adrSpaceShow**

adrSpaceRAMReserve( )

NAME **adrSpaceRAMReserve( )** – reserve memory from the RAM pool

SYNOPSIS

```
PHYS_ADDR adrSpaceRAMReserve
    (
    PHYS_ADDR startAddr,  /* start adress of RAM memory block */
    UINT      ramSize     /* in bytes */
    )
```

DESCRIPTION This routine reserves memory from the system global RAM page pool. If the *startAddr* parameter is NONE, the pages will be allocated from the free pages available in the RAM pool. If the *startAddr* parameter is not NONE this routine will attempt to get the pages at that address. If the pages are not available, the routine fails.

The length and start address are expected to be MMU page aligned. The smallest amount of space that can be reserved is one page.

RETURNS physical address on success, NONE otherwise.

ERRNO Possible errnos generated by this routine include:

S_adrSpaceLib_SIZE_IS_INVALID
ramSize is less than a page.

S_adrSpaceLib_PARAMETER_NOT_ALIGNED
start or end address is not page size aligned.

SEE ALSO **adrSpaceLib**, **adrSpaceShow**

adrSpaceShow( )

NAME

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

SYNOPSIS

```
STATUS adrSpaceShow
    (
    UINT level  /* verbosity level: 0 = summary, 1 = details */
    )
```

DESCRIPTION

This routine displays information about various address space regions managed by the address space library, **adrSpaceLib**. When the parameter, *level*, is not 0, more detailed information is displayed.

The information is displayed under the following headings:

RAM Physical Address Space Info
: Displays information about the physical address space of the system RAM.

User Region Info
: Displays information about the user virtual address space for the creation of RTP's, shared libraries and shared data regions.

Kernel Region Info
: Displays information about the kernel virtual space, including memory-mapped IO space.

EXAMPLE

```
-> adrSpaceShow 1

RAM Physical Address Space Info:
-------------------------------
        Allocation unit size:           0x1000
        Total number of units:          131072      (536870912 bytes)
        Number of allocated units:      96127       (393736192 bytes)
        Largest contiguous free block:  143134720
        Number of free units:           34945       (143134720 bytes)
               1 block(s) of 0x08881000 bytes  (0x1777f000-0x1fffffff)

User Region (RTP/SL/SD) Virtual Space Info:
-------------------------------------------
        Allocation unit size:           0x1000
        Total number of units:          720896      (2952790016 bytes)
        Number of allocated units:      0           (0 bytes)
        Largest contiguous free block:  1610612736
        Number of free units:           720896      (2952790016 bytes)
               1 block(s) of 0x50000000 bytes  (0x90000000-0xdfffffff)
               1 block(s) of 0x60000000 bytes  (0x20000000-0x7fffffff)

Kernel Region Virtual Space Info:
---------------------------------
        Allocation unit size:           0x1000
        Number reserved of units:       327680     (1342177280 bytes)
               1 block(s) of 0x20000000 bytes  (0x00000000-0x1fffffff)
               1 block(s) of 0x10000000 bytes  (0x80000000-0x8fffffff)
value = 0 = 0x0
```

RETURNS **OK**, always

ERRNO Not Available

SEE ALSO **adrSpaceShow**, **adrSpaceLib**

adrSpaceVirtReserve( )

NAME **adrSpaceVirtReserve( )** – reserve memory from the virtual space

SYNOPSIS

```
VIRT_ADDR adrSpaceVirtReserve
    (
    VIRT_ADDR startAddr,  /* start adress of virtual memory block */
    UINT      numPages    /* in bytes */
    )
```

DESCRIPTION This routine reserves virtual memory pages from the user region. If the *startAddr* parameter is NONE, the pages will be allocated from the free pages available in the region. If the *startAddr* parameter is not NONE this routine will attempt to get the pages at that address. If the pages are not available, the routine fails.

The length and start address are expected to be MMU page aligned. The smallest amount of space that can be reserved is one page.

RETURNS virtual address on success, NONE otherwise.

ERRNO Possible errnos generated by this routine include:

S_adrSpaceLib_SIZE_IS_INVALID
ramSize is less than a page.

S_adrSpaceLib_PARAMETER_NOT_ALIGNED
start or end address is not page size aligned.

SEE ALSO **adrSpaceLib**, **adrSpaceShow**

aimCacheInit( )

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

SYNOPSIS

```
STATUS aimCacheInit
    (
    CACHECONFIG * cacheConfig
    )
```

DESCRIPTION This routine is called by the bsp from 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.

RETURNS **ERROR** if an invalid cache operation is requested, otherwise **OK**.

ERRNO N/A

SEE ALSO **aimCacheLib**

aimFppLibInit( )

NAME **aimFppLibInit( )** – Initialize the AIM FPU library

SYNOPSIS

```
void aimFppLibInit
    (
    void
    )
```

DESCRIPTION Initialize the AIM FPU library.

RETURNS N/A

ERRNO

SEE ALSO **aimFppLib**

aimMmuLibInit( )

NAME **aimMmuLibInit( )** – initialize the AIM

SYNOPSIS

```
STATUS aimMmuLibInit (void)
```

DESCRIPTION **aimMmuLibInit( )** performs AIM-specific initialization of the MMU subsystem.

If _WRS_NONGLOBAL_NULL_PAGE is defined, the null page pte entries are forced to be non-global. This avoids a potential conflict within the MMU between a virtual address that is (non-globally) mapped in one vm context, then accessed in a different vm context that does not have the address mapped.

RETURNS **OK**, or **ERROR** if any problems are encountered during init.

ERRNO

SEE ALSO **aimMmuLib**

aioShow( )

NAME **aioShow( )** – show AIO requests

SYNOPSIS

```
STATUS aioShow
    (
    int drvNum  /* drv num to show (IGNORED) */
    )
```

DESCRIPTION This routine displays the outstanding AIO requests.

CAVEAT The *drvNum* parameter is not used.

RETURNS **OK**, always.

ERRNO N/A.

SEE ALSO **aioPxShow**

aioSysInit( )

NAME **aioSysInit( )** – initialize the AIO system driver

SYNOPSIS

```
STATUS aioSysInit
    (
    int numTasks,      /* number of system tasks */
    int taskPrio,      /* AIO task priority */
    int taskStackSize  /* AIO task stack size */
    )
```

DESCRIPTION This routine initializes the AIO system driver. It should be called once after the AIO library has been initialized. It spawns *numTasks* system I/O tasks to be executed at *taskPrio* priority level, with a stack size of *taskStackSize*. It also starts the wait task and sets the system driver as the default driver for AIO. If *numTasks*, *taskPrio*, or *taskStackSize* is 0, a default value (**AIO_IO_TASKS_DFLT**, **AIO_IO_PRIO_DFLT**, or **AIO_IO_STACK_DFLT**, respectively) is used.

RETURNS **OK** if successful, otherwise **ERROR**.

ERRNO N/A.

SEE ALSO **aioSysDrv**

aio_cancel( )

NAME **aio_cancel( )** – cancel an asynchronous I/O request (POSIX)

SYNOPSIS

```
int aio_cancel
    (
    int            fildes,  /* file descriptor */
    struct aiocb * pAiocb   /* AIO control block */
    )
```

DESCRIPTION This routine attempts to cancel one or more asynchronous I/O request(s) currently outstanding against the file descriptor *fildes*. *pAiocb* points to the asynchronous I/O control block for a particular request to be cancelled. If *pAiocb* is **NULL**, all outstanding cancelable asynchronous I/O requests associated with *fildes* are cancelled.

Normal signal delivery occurs for AIO operations that are successfully cancelled. If there are requests that cannot be cancelled, then the normal asynchronous completion process takes place for those requests when they complete.

Operations that are cancelled successfully have a return status of -1 and an error status of **ECANCELED**.

RETURNS

AIO_CANCELED if requested operations were cancelled,
AIO_NOTCANCELED if at least one operation could not be cancelled,
AIO_ALLDONE if all operations have already completed, or
ERROR if an error occurred.

ERRNO

EBADF
Invalid, or closed file descriptor.

SEE ALSO

aioPxLib, **aio_return( )**, **aio_error( )**

aio_error( )

NAME

aio_error( ) – retrieve error status of asynchronous I/O operation (POSIX)

SYNOPSIS

```
int aio_error
    (
    const struct aiocb * pAiocb  /* AIO control block */
    )
```

DESCRIPTION

This routine returns the error status associated with the I/O operation specified by *pAiocb*. If the operation is not yet completed, the error status will be **EINPROGRESS**.

RETURNS

EINPROGRESS if the AIO operation has not yet completed,
OK if the AIO operation completed successfully,
the error status if the AIO operation failed,
otherwise **ERROR**.

ERRNO

EINVAL

SEE ALSO

aioPxLib

aio_fsync( )

NAME **aio_fsync( )** – asynchronous file synchronization (POSIX)

SYNOPSIS

```
int aio_fsync
    (
    int             op,      /* operation */
    struct aiocb * pAiocb  /* AIO control block */
    )
```

DESCRIPTION This routine asynchronously forces all I/O operations associated with the file, indicated by **aio_fildes**, queued at the time **aio_fsync( )** is called to the synchronized I/O completion state. **aio_fsync( )** returns when the synchronization request has be initiated or queued to the file or device.

The value of *op* is either **O_DSYNC** or **O_SYNC**.

If the call fails, the outstanding I/O operations are not guaranteed to have completed. If it succeeds, only the I/O that was queued at the time of the call is guaranteed to the relevant completion state.

The **aio_sigevent** member of the *pAiocb* defines an optional signal to be generated on completion of **aio_fsync( )**.

RETURNS **OK** if queued successfully, otherwise **ERROR**.

ERRNO **EINVAL**
EBADF

SEE ALSO **aioPxLib**, **aio_error( )**, **aio_return( )**

aio_read( )

NAME **aio_read( )** – initiate an asynchronous read (POSIX)

SYNOPSIS

```
int aio_read
    (
    struct aiocb * pAiocb  /* AIO control block */
    )
```

DESCRIPTION This routine asynchronously reads data based on the following parameters specified by members of the AIO control structure *pAiocb*. It reads **aio_nbytes** bytes of data from the file **aio_fildes** into the buffer **aio_buf**.

The requested operation takes place at the absolute position in the file as specified by **aio_offset**.

aio_reqprio can be used to lower the priority of the AIO request; if this parameter is nonzero, the priority of the AIO request is **aio_reqprio** lower than the calling task priority.

The call returns when the read request has been initiated or queued to the device. **aio_error( )** can be used to determine the error status and of the AIO operation. On completion, **aio_return( )** can be used to determine the return status.

aio_sigevent defines the signal to be generated on completion of the read request. If this value is zero, no signal is generated.

RETURNS

OK if the read queued successfully, otherwise **ERROR**.

ERRNO

EBADF
EINVAL

SEE ALSO

aioPxLib, **aio_error( )**, **aio_return( )**, **read( )**

aio_return( )

NAME

aio_return( ) – retrieve return status of asynchronous I/O operation (POSIX)

SYNOPSIS

```
ssize_t aio_return
    (
    struct aiocb * pAiocb  /* AIO control block */
    )
```

DESCRIPTION

This routine returns the return status associated with the I/O operation specified by *pAiocb*. The return status for an AIO operation is the value that would be returned by the corresponding **read( )**, **write( )**, or **fsync( )** call. **aio_return( )** may be called only after the AIO operation has completed (**aio_error( )** returns a valid error code--not **EINPROGRESS**). Furthermore, **aio_return( )** may be called only once; subsequent calls will fail.

RETURNS

The return status of the completed AIO request, or **ERROR**.

ERRNO

EINVAL
EINPROGRESS

SEE ALSO

aioPxLib

aio_suspend( )

NAME

aio_suspend( ) – wait for asynchronous I/O request(s) (POSIX)

SYNOPSIS

```
int aio_suspend
    (
    const struct aiocb *const list[],  /* AIO requests */
    int                       nEnt,    /* number of requests */
    const struct timespec *   timeout  /* wait timeout */
    )
```

DESCRIPTION

This routine suspends the caller until one of the following occurs:

- at least one of the previously submitted asynchronous I/O operations referenced by *list* has completed,
- a signal interrupts the function, or
- the time interval specified by *timeout* has passed (if *timeout* is not **NULL**).

RETURNS

OK if an AIO request completes, otherwise **ERROR**.

ERRNO

EAGAIN
EINTR

SEE ALSO

aioPxLib

aio_write( )

NAME

aio_write( ) – initiate an asynchronous write (POSIX)

SYNOPSIS

```
int aio_write
    (
    struct aiocb * pAiocb  /* AIO control block */
    )
```

DESCRIPTION

This routine asynchronously writes data based on the following parameters specified by members of the AIO control structure *pAiocb*. It writes **aio_nbytes** of data to the file **aio_fildes** from the buffer **aio_buf**.

The requested operation takes place at the absolute position in the file as specified by **aio_offset**.

aio_reqprio can be used to lower the priority of the AIO request; if this parameter is nonzero, the priority of the AIO request is **aio_reqprio** lower than the calling task priority.

The call returns when the write request has been initiated or queued to the device. **aio_error( )** can be used to determine the error status and of the AIO operation. On completion, **aio_return( )** can be used to determine the return status.

aio_sigevent defines the signal to be generated on completion of the write request. If this value is zero, no signal is generated.

RETURNS **OK** if write queued successfully, otherwise **ERROR**.

ERRNO **EBADF**
EINVAL

SEE ALSO **aioPxLib**, **aio_error( )**, **aio_return( )**, **write( )**

alarm( )

NAME **alarm( )** – set an alarm clock for delivery of a signal

SYNOPSIS

```
unsigned int alarm
    (
    unsigned int secs
    )
```

DESCRIPTION This routine arranges for a **SIGALRM** signal to be delivered to the calling task after *secs* seconds.

If *secs* is zero, no new alarm is scheduled. In all cases, any previously set alarm is cancelled.

RETURNS Time remaining until a previously scheduled alarm was due to be delivered, zero if there was no previous alarm, or **ERROR** in case of an error.

ERRNO **EINVAL**

ENOSYS

EAGAIN

S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO **timerLib**

anRegister( )

NAME
anRegister( ) – register with the VxBus subsystem

SYNOPSIS
```
void anRegister(void)
```

DESCRIPTION
This routine registers the AN983 driver with VxBus as a child of the PCI bus type.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
an983VxbEnd

asinf( )

NAME
asinf( ) – compute an arc sine (ANSI)

SYNOPSIS
```
float asinf
    (
    float x  /* number between -1 and 1 */
    )
```

DESCRIPTION
This routine computes the arc sine of *x* in single precision. If *x* is the sine of an angle *T*, this function returns *T*.

RETURNS
The single-precision arc sine of *x* in the range -pi/2 to pi/2 radians.

ERRNO
Not Available

SEE ALSO
mathALib

atan2f( )

NAME
atan2f( ) – compute the arc tangent of y/x (ANSI)

SYNOPSIS
```
float atan2f
    (
    float y,  /* numerator */
    float x   /* denominator */
    )
```

DESCRIPTION This routine returns the principal value of the arc tangent of *y*/*x* in single precision.

RETURNS The single-precision arc tangent of *y*/*x* in the range -pi to pi.

ERRNO Not Available

SEE ALSO **mathALib**

atanf( )

NAME **atanf( )** – compute an arc tangent (ANSI)

SYNOPSIS

```
float atanf
    (
    float x  /* tangent of an angle */
    )
```

DESCRIPTION This routine computes the arc tangent of *x* in single precision. If *x* is the tangent of an angle *T*, this function returns *T* (in radians).

RETURNS The single-precision arc tangent of *x* in the range -pi/2 to pi/2.

ERRNO Not Available

SEE ALSO **mathALib**

atapiParamsPrint( )

NAME **atapiParamsPrint( )** – Print the drive parameters.

SYNOPSIS

```
void atapiParamsPrint
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This user callable routine will read the current parameters from the corresponding drive and will print the specified range of parameters on the console.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

attrib( )

NAME **attrib( )** – modify MS-DOS file attributes on a file or directory

SYNOPSIS

```
STATUS attrib
    (
    const char * fileName,  /* file or dir name on which to change flags */
    const char * attr       /* flag settings to change */
    )
```

DESCRIPTION This function provides means for the user to modify the attributes of a single file or directory. There are four attribute flags which may be modified: "Archive", "System", "Hidden" and "Read-only". Among these flags, only "Read-only" has a meaning in VxWorks, namely, read-only files can not be modified deleted or renamed.

The *attr* argument string may contain must start with either "+" or "-", meaning the attribute flags which will follow should be either set or cleared. After "+" or "-" any of these four letter will signify their respective attribute flags - "A", "S", "H" and "R".

For example, to write-protect a particular file and flag that it is a system file:

```
-> attrib( "bootrom.sys", "+RS")
```

RETURNS **OK**, or **ERROR** if the file can not be opened.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dosFsLib**, the VxWorks programmer guides.

b( )

NAME

b( ) – set or display breakpoints

SYNOPSIS

```
STATUS b
    (
    INSTR * addr,           /* breakpoint addr, or 0 to display */
    int     taskNameOrId,   /* task affected; 0 means all tasks      */
    int     count,          /* number of passes before hit   */
    BOOL    quiet           /* TRUE = don't print debugging info,  */
                            /* FALSE = print debugging info */
    )
```

DESCRIPTION

This routine sets or displays breakpoints. To display the list of currently active breakpoints, call **b( )** without arguments:

```
    -> b
```

The list shows the address, task, context type, action, notification status and pass count of each breakpoint. Temporary breakpoints inserted by **so( )** and **cret( )** are also indicated.

To set a breakpoint with **b( )**, include the address, which can be specified numerically or symbolically with an optional offset. The other arguments are optional:

```
    -> b addr[,task[,count[,quiet]]]
```

If *task* is zero or omitted, the breakpoint will apply to all breakable tasks. If *count* is zero or omitted, the breakpoint will occur every time it is hit. If *count* is specified, the break will not occur until the *count* +1th time an eligible task hits the breakpoint (i.e., the breakpoint is ignored the first *count* times it is hit).

If *quiet* is specified, debugging information destined for the console will be suppressed when the breakpoint is hit. This option is included for use by external source code debuggers that handle the breakpoint user interface themselves.

Individual tasks can be unbreakable, in which case breakpoints that otherwise would apply to a task are ignored. Tasks can be spawned unbreakable by specifying the task option **VX_UNBREAKABLE**. Tasks can also be set unbreakable or breakable by resetting **VX_UNBREAKABLE** with the routine **taskOptionsSet( )**.

RETURNS

OK, or **ERROR** if *addr* is illegal or the breakpoint table is full.

ERRNO

N/A

SEE ALSO

dbgLib, **bd( )**, **taskOptionsSet( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

bcmp( )

NAME

bcmp( ) – compare one buffer to another

SYNOPSIS

```
int bcmp
    (
    FAST char *buf1,  /* pointer to first buffer    */
    FAST char *buf2,  /* pointer to second buffer   */
    FAST int  nbytes  /* number of bytes to compare */
    )
```

DESCRIPTION

This routine compares the first *nbytes* characters of *buf1* to *buf2*.

RETURNS

- 0 if the first *nbytes* of *buf1* and *buf2* are identical,
- less than 0 if *buf1* is less than *buf2*, or
- greater than 0 if *buf1* is greater than *buf2*.

ERRNO

N/A

SEE ALSO

bLib

bcopy( )

NAME

bcopy( ) – copy one buffer to another

SYNOPSIS

```
void bcopy
    (
    const char *source,       /* pointer to source buffer      */
    char       *destination,  /* pointer to destination buffer */
    int        nbytes         /* number of bytes to copy       */
    )
```

DESCRIPTION

This routine copies the first *nbytes* characters from *source* to *destination*. Overlapping buffers are handled correctly. Copying is done in the most efficient way possible, which may include long-word, or even multiple-long-word moves on some architectures. In general, the copy will be significantly faster if both buffers are long-word aligned. (For copying that is restricted to byte, word, or long-word moves, see the manual entries for **bcopyBytes( )**, **bcopyWords( )**, and **bcopyLongs( )**.)

RETURNS

N/A

ERRNO

N/A

SEE ALSO **bLib**, **bcopyBytes( )**, **bcopyWords( )**, **bcopyLongs( )**

bcopyBytes( )

NAME **bcopyBytes( )** – copy one buffer to another one byte at a time

SYNOPSIS

```
void bcopyBytes
    (
    char *source,       /* pointer to source buffer      */
    char *destination,  /* pointer to destination buffer */
    int  nbytes         /* number of bytes to copy       */
    )
```

DESCRIPTION This routine copies the first *nbytes* characters from *source* to *destination* one byte at a time. This may be desirable if a buffer can only be accessed with byte instructions, as in certain byte-wide memory-mapped peripherals.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **bcopy( )**

bcopyLongs( )

NAME **bcopyLongs( )** – copy one buffer to another one long word at a time

SYNOPSIS

```
void bcopyLongs
    (
    char *source,       /* pointer to source buffer      */
    char *destination,  /* pointer to destination buffer */
    int  nlongs         /* number of longs to copy       */
    )
```

DESCRIPTION This routine copies the first *nlongs* characters from *source* to *destination* one long word at a time. This may be desirable if a buffer can only be accessed with long instructions, as in certain long-word-wide memory-mapped peripherals. The source and destination must be long-aligned.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **bcopy( )**

bcopyWords( )

NAME **bcopyWords( )** – copy one buffer to another one word at a time

SYNOPSIS

```
void bcopyWords
    (
    char *source,       /* pointer to source buffer      */
    char *destination,  /* pointer to destination buffer */
    int  nwords         /* number of words to copy       */
    )
```

DESCRIPTION This routine copies the first *nwords* words from *source* to *destination* one word at a time. This may be desirable if a buffer can only be accessed with word instructions, as in certain word-wide memory-mapped peripherals. The source and destination must be word-aligned.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **bcopy( )**

bd( )

NAME **bd( )** – delete a breakpoint

SYNOPSIS

```
STATUS bd
    (
    INSTR * addr,          /* address of breakpoint to delete      */
    int     taskNameOrId   /* task affected; 0 means all tasks     */
    )
```

DESCRIPTION This routine deletes a specified breakpoint, based on its address.

To execute, enter:

```
    -> bd addr [,task]
```

If *task* is omitted or zero, the breakpoint will be removed for all tasks. If the breakpoint applies to all tasks, removing it for only a single task will be ineffective. It must be removed for all tasks and then set for just those tasks desired. Temporary breakpoints inserted by the routines **so( )** or **cret( )** can also be deleted.

RETURNS

OK, or **ERROR** if there is no breakpoint at the specified address.

ERRNO

N/A

SEE ALSO

dbgLib, **b( )**, **bdall( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

bdall( )

NAME

bdall( ) – delete all breakpoints

SYNOPSIS

```
STATUS bdall
    (
    int taskNameOrId  /* task affected; 0 means all tasks     */
    )
```

DESCRIPTION

This routine removes all breakpoints.

To execute, enter:

```
    -> bdall [task]
```

If *task* is specified, all breakpoints that apply to that task are removed. If *task* is omitted, all breakpoints for all tasks are removed. Temporary breakpoints inserted by **so( )** or **cret( )** are not deleted; use **bd( )** instead.

RETURNS

OK, or **ERROR** if *task* cannot be found.

ERRNO

N/A

SEE ALSO

dbgLib, **b( )**, **bd( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

bfStrSearch( )

NAME **bfStrSearch( )** – Search using the Brute Force algorithm

SYNOPSIS

```
char * bfStrSearch
    (
    char * pattern,       /* pattern to search for */
    int    patternLen,    /* length of the pattern */
    char * buffer,        /* text buffer to search in */
    int    bufferLen,     /* length of the text buffer */
    BOOL   caseSensitive  /* case-sensitive search? */
    )
```

DESCRIPTION The Brute Force algorithm is the simplest string search algorithm. It performs comparisons between a character in the pattern and a character in the text buffer from left to right. After each attempt it shifts the pattern by one position to the right.

The Brute Force algorithm requires no pre-processing and no extra space. It has a O(Pattern Length x Text Buffer Length) worst-case time complexity.

RETURNS A pointer to the located pattern, or a **NULL** pointer if the pattern is not found

ERRNO Not Available

SEE ALSO **strSearchLib**

bfill( )

NAME **bfill( )** – fill a buffer with a specified character

SYNOPSIS

```
void bfill
    (
    FAST char *buf,    /* pointer to buffer              */
    int       nbytes,  /* number of bytes to fill        */
    FAST int  ch       /* char with which to fill buffer */
    )
```

DESCRIPTION This routine fills the first *nbytes* characters of a buffer with the character *ch*. Filling is done in the most efficient way possible, which may be long-word, or even multiple-long-word stores, on some architectures. In general, the fill will be significantly faster if the buffer is long-word aligned. (For filling that is restricted to byte stores, see the manual entry for **bfillBytes( )**.)

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **bfillBytes( )**

bfillBytes( )

NAME **bfillBytes( )** – fill buffer with a specified character one byte at a time

SYNOPSIS

```
void bfillBytes
    (
    FAST char *buf,    /* pointer to buffer              */
    int       nbytes,  /* number of bytes to fill        */
    FAST int  ch       /* char with which to fill buffer */
    )
```

DESCRIPTION This routine fills the first *nbytes* characters of the specified buffer with the character *ch* one byte at a time. This may be desirable if a buffer can only be accessed with byte instructions, as in certain byte-wide memory-mapped peripherals.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **bfill( )**

bh( )

NAME **bh( )** – set a hardware breakpoint

SYNOPSIS

```
STATUS bh
    (
    INSTR * addr,          /* where to set breakpoint, or */
                           /* 0 = display all breakpoints */
    int     access,        /* access type (arch dependant) */
    int     taskNameOrId,  /* task affected; 0 means all tasks     */
    int     count,         /* number of passes before hit          */
    BOOL    quiet          /* TRUE = don't print debugging info,  */
                           /* FALSE = print debugging info */
    )
```

DESCRIPTION This routine is used to set a hardware breakpoint. If the architecture allows it, this function adds the breakpoint to the list of breakpoints and set the hardware breakpoint register(s). For more information, see the manual entry for **b( )**.

NOTE The types of hardware breakpoints vary with the architectures. Generally, a hardware breakpoint can be a data breakpoint or an instruction breakpoint.

RETURNS **OK**, or **ERROR** if *addr* is illegal or the hardware breakpoint table is full.

ERRNO N/A

SEE ALSO **dbgLib**, **b( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

binvert( )

NAME **binvert( )** – invert the order of bytes in a buffer

SYNOPSIS

```
void binvert
    (
    FAST char * buf,    /* pointer to buffer to invert  */
    int         nbytes  /* number of bytes in buffer    */
    )
```

DESCRIPTION This routine inverts an entire buffer, byte by byte. For example, the buffer {1, 2, 3, 4, 5} would become {5, 4, 3, 2, 1}.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**

bmsStrSearch( )

NAME **bmsStrSearch( )** – Search using the Boyer-Moore-Sunday (Quick Search) algorithm

SYNOPSIS

```
char * bmsStrSearch
    (
    char * pattern,       /* pattern to search for */
    int    patternLen,    /* length of the pattern */
    char * buffer,        /* text buffer to search in */
    int    bufferLen,     /* length of the text buffer */
    BOOL   caseSensitive  /* case-sensitive search? */
    )
```

DESCRIPTION

The Boyer-Moore-Sunday algorithm is a more efficient simplification of the Boyer-Moore algorithm. It performs comparisons between a character in the pattern and a character in the text buffer from left to right. After each mismatch it uses bad character heuristic to shift the pattern to the right. For more details on the algorithm, refer to "A Very Fast Substring Search Algorithm", Daniel M. Sunday, Communications of the ACM, Vol. 33 No. 8, August 1990, pp. 132-142.

It has a O(Pattern Length x Text Buffer Length) worst-case time complexity. But empirical results have shown that this algorithm is one of the fastest in practice.

RETURNS

A pointer to the located pattern, or a **NULL** pointer if the pattern is not found

ERRNO

Not Available

SEE ALSO

strSearchLib

bmtPhyRegister( )

NAME

bmtPhyRegister( ) – register with the VxBus subsystem

SYNOPSIS

```
void bmtPhyRegister(void)
```

DESCRIPTION

This routine registers the BCM52xx driver with VxBus as a child of the MII bus type.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

bcm52xxPhy

bootBpAnchorExtract( )

NAME

bootBpAnchorExtract( ) – extract a backplane address from a device field

SYNOPSIS

```
STATUS bootBpAnchorExtract
    (
    char *string,       /* string containing adrs field */
    char **pAnchorAdrs  /* pointer where to return anchor address */
    )
```

DESCRIPTION

This routine extracts the optional backplane anchor address field from a boot device field. The anchor can be specified for the backplane driver by appending to the device name (i.e., "bp") an equal sign (=) and the address in hexadecimal. For example, the "boot device" field of the boot parameters could be specified as:

```
    boot device: bp=800000
```

In this case, the backplane anchor address would be at address 0x800000, instead of the default specified in **config.h**.

This routine picks off the optional trailing anchor address by replacing the equal sign (=) in the specified string with an EOS and then scanning the remainder as a hex number. This number, the anchor address, is returned via the *pAnchorAdrs* pointer.

RETURNS

1 if the anchor address in *string* is specified correctly,
0 if the anchor address in *string* is not specified, or
-1 if an invalid anchor address is specified in *string*.

ERRNO

Not Available

SEE ALSO

bootParseLib

bootChange( )

NAME

bootChange( ) – change the boot line

SYNOPSIS

```
void bootChange (void)
```

DESCRIPTION

This command changes the boot line used in the boot ROMs. This is useful during a remote login session. After changing the boot parameters, you can reboot the target with the **reboot( )** command, and then terminate your login (~.) and remotely log in again. As soon as the system has rebooted, you will be logged in again.

This command stores the new boot line in non-volatile RAM, if the target has it.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, the VxWorks programmer guides.

bootLeaseExtract( )

NAME **bootLeaseExtract( )** – extract the lease information from an Internet address

SYNOPSIS

```
int bootLeaseExtract
    (
    char   *string,       /* string containing addr field */
    u_long *pLeaseLen,    /* pointer to storage for lease duration */
    u_long *pLeaseStart  /* pointer to storage for lease origin */
    )
```

DESCRIPTION This routine extracts the optional lease duration and lease origin fields from an Internet address field for use with DHCP. The lease duration can be specified by appending a colon and the lease duration to the netmask field. For example, the "inet on ethernet" field of the boot parameters could be specified as:

```
inet on ethernet: 90.1.0.1:ffff0000:1000
```

If no netmask is specified, the contents of the field could be:

```
inet on ethernet: 90.1.0.1::ffffffff
```

In the first case, the lease duration for the address is 1000 seconds. The second case indicates an infinite lease, and does not specify a netmask for the address. At the beginning of the boot process, the value of the lease duration field is used to specify the requested lease duration. If the field not included, the value of **DHCP_DEFAULT_LEASE** is used instead.

The lease origin is specified with the same format as the lease duration, but is added during the boot process. The presence of the lease origin field distinguishes addresses assigned by a DHCP server from addresses entered manually. Addresses assigned by a DHCP server may be replaced if the bootstrap loader uses DHCP to obtain configuration parameters. The value of the lease origin field at the beginning of the boot process is ignored.

This routine extracts the optional lease duration by replacing the preceding colon in the specified string with an EOS and then scanning the remainder as a number. The lease duration and lease origin values are returned via the *pLeaseLen* and *pLeaseStart* pointers, if those parameters are not **NULL**.

RETURNS
2 if both lease values are specified correctly in *string*, or
-2 if one of the two values is specified incorrectly.
If only the lease duration is found, it returns:
1 if the lease duration in *string* is specified correctly,
0 if the lease duration is not specified in *string*, or
-1 if an invalid lease duration is specified in *string*.

ERRNO
Not Available

SEE ALSO
bootParseLib

bootNetmaskExtract( )

NAME
bootNetmaskExtract( ) – extract the net mask field from an Internet address

SYNOPSIS
```
STATUS bootNetmaskExtract
    (
    char *string,   /* string containing addr field */
    int  *pNetmask  /* pointer where to return net mask */
    )
```

DESCRIPTION
This routine extracts the optional subnet mask field from an Internet address field. Subnet masks can be specified for an Internet interface by appending to the Internet address a colon and the net mask in hexadecimal. For example, the "inet on ethernet" field of the boot parameters could be specified as:

```
    inet on ethernet: 90.1.0.1:ffff0000
```

In this case, the network portion of the address (normally just 90) is extended by the subnet mask (to 90.1). This routine extracts the optional trailing subnet mask by replacing the colon in the specified string with an EOS and then scanning the remainder as a hex number. This number, the net mask, is returned via the *pNetmask* pointer.

This routine also handles an empty netmask field used as a placeholder for the lease duration field (see **bootLeaseExtract( )**). In that case, the colon separator is replaced with an EOS and the value of netmask is set to 0.

RETURNS
1 if the subnet mask in *string* is specified correctly,
0 if the subnet mask in *string* is not specified, or
-1 if an invalid subnet mask is specified in *string*.

ERRNO
Not Available

SEE ALSO **bootParseLib**

bootParamsPrompt( )

NAME **bootParamsPrompt( )** – prompt for boot line parameters

SYNOPSIS

```
void bootParamsPrompt
    (
    char *string  /* default boot line */
    )
```

DESCRIPTION This routine displays the current value of each boot parameter and prompts the user for a new value. Typing a RETURN leaves the parameter unchanged. Typing a period (.) clears the parameter.

The parameter *string* holds the initial values. The new boot line is copied over *string*. If there are no initial values, *string* is empty on entry.

RETURNS N/A

ERRNO Not Available

SEE ALSO **bootLib**

bootParamsShow( )

NAME **bootParamsShow( )** – display boot line parameters

SYNOPSIS

```
void bootParamsShow
    (
    char *paramString  /* boot parameter string */
    )
```

DESCRIPTION This routine displays the boot parameters in the specified boot string one parameter per line.

RETURNS N/A

ERRNO Not Available

SEE ALSO **bootLib**

bootStringToStruct( )

NAME **bootStringToStruct( )** – interpret the boot parameters from the boot line

SYNOPSIS

```
char * bootStringToStruct
    (
    char *             bootString,  /* boot line to be parsed */
    FAST BOOT_PARAMS * pBootParams  /* where to return parsed boot line */
    )
```

DESCRIPTION This routine parses the ASCII string and returns the values into the provided parameters.

For a description of the format of the boot line, see the manual entry for **bootLib**

RETURNS A pointer to the last character successfully parsed plus one (points to EOS, if **OK**). The entire boot line is parsed.

ERRNO Not Available

SEE ALSO **bootParseLib**

bootStringToStructAdd( )

NAME **bootStringToStructAdd( )** – interpret the boot parameters from the boot line

SYNOPSIS

```
char * bootStringToStructAdd
    (
    char *             bootString,  /* boot line to be parsed */
    FAST BOOT_PARAMS * pBootParams  /* where to return parsed boot line */
    )
```

DESCRIPTION This routine parses the ASCII string *bootString* and returns the values into the provided parameters *pBootParams*. The fields of *pBootParams* may be previously set to default values.

For a description of the format of the boot line, see the manual entry for **bootLib**

RETURNS A pointer to the last character successfully parsed plus one (points to EOS, if **OK**). The entire boot line is parsed.

ERRNO Not Available

SEE ALSO **bootParseLib**

bootStructToString( )

NAME **bootStructToString( )** – construct a boot line

SYNOPSIS

```
STATUS bootStructToString
    (
    char             *paramString,  /* where to return the encoded boot line
*/
    FAST BOOT_PARAMS *pBootParams   /* boot line structure to be encoded */
    )
```

DESCRIPTION This routine encodes a boot line using the specified boot parameters.

For a description of the format of the boot line, see the manual entry for **bootLib**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **bootLib**

bswap( )

NAME **bswap( )** – swap buffers

SYNOPSIS

```
void bswap
    (
    FAST char * buf1,   /* pointer to first buffer  */
    FAST char * buf2,   /* pointer to second buffer */
    FAST int    nbytes  /* number of bytes to swap  */
    )
```

DESCRIPTION This routine exchanges the first *nbytes* of the two specified buffers.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**

bzero( )

NAME **bzero( )** – zero out a buffer

SYNOPSIS

```
void bzero
    (
    char * buffer,  /* buffer to be zeroed       */
    int    nbytes   /* number of bytes in buffer */
    )
```

DESCRIPTION This routine fills the first *nbytes* characters of the specified buffer with 0.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**

c( )

NAME **c( )** – continue from a breakpoint

SYNOPSIS

```
STATUS c
    (
    int     taskNameOrId,  /* task that should proceed from breakpoint     */
    INSTR * addr,          /* address to continue at; 0 = next instruction */
    INSTR * addr1          /* address for npc; 0 = instruction next to pc */
    )
```

DESCRIPTION This routine continues the execution of a task that has stopped at a breakpoint.

To execute, enter:

```
    -> c [task [,addr[,addr1]]]
```

If *task* is omitted or zero, the last task referenced is assumed. If *addr* is non-zero, the program counter is changed to *addr*; if *addr1* is non-zero, the next program counter is changed to *addr1*, and the task is continued.

CAVEAT When a task is continued, **c( )** does not distinguish between a stopped task or a task stopped by the debugger. Therefore, its use should be restricted to only those tasks being debugged.

NOTE The next program counter, *addr1*, is currently supported only by SPARC.

RETURNS **OK**, or **ERROR** if the specified task does not exist.

ERRNO N/A

SEE ALSO **dbgLib**, **s( )**, **cret( )**, **tr( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

cacheArchClearEntry( )

NAME **cacheArchClearEntry( )** – clear an entry from a cache (68K, x86)

SYNOPSIS

```
STATUS cacheArchClearEntry
    (
    CACHE_TYPE cache,   /* cache to clear entry for */
    void *     address  /* entry to clear */
    )
```

DESCRIPTION This routine clears a specified entry from the specified cache.

For 68040 processors, this routine clears the cache line from the cache in which the cache entry resides.

For the MC68060 processor, when the instruction cache is cleared (invalidated) the branch cache is also invalidated by the hardware. One line in the branch cache cannot be invalidated so each time the branch cache is entirely invalidated.

For 386 family processors do not have a cache, thus it does nothing. The 486, P5(Pentium), and P6(PentiumPro, II, III) family processors do have a cache but does not support a line by line cache control, thus it performs WBINVD instruction. The P7(Pentium4) family processors support the line by line cache control with CLFLUSH instruction, thus flushes the specified cache line.

RETURNS **OK**, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO Not Available

SEE ALSO **cacheArchLib**

cacheArchLibInit( )

NAME

cacheArchLibInit( ) – initialize the cache library

SYNOPSIS

```
STATUS cacheArchLibInit
    (
    CACHE_MODE instMode,  /* instruction cache mode */
    CACHE_MODE dataMode   /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the cache library for the following processor cache families: Motorola 68K, Intel x86, PowerPC ARM, and the Solaris and Windows simulators. It initializes the function pointers and configures the caches to the specified cache modes.

68K PROCESSORS The caching modes vary for members of the 68K processor family:

68020:	**CACHE_WRITETHROUGH**	(instruction cache only)
68030:	**CACHE_WRITETHROUGH**	
	CACHE_BURST_ENABLE	
	CACHE_BURST_DISABLE	
	CACHE_WRITEALLOCATE	(data cache only)
	CACHE_NO_WRITEALLOCATE	(data cache only)
68040:	**CACHE_WRITETHROUGH**	
	CACHE_COPYBACK	(data cache only)
	CACHE_INH_SERIAL	(data cache only)
	CACHE_INH_NONSERIAL	(data cache only)
	CACHE_BURST_ENABLE	(data cache only)
	CACHE_NO_WRITEALLOCATE	(data cache only)
68060:	**CACHE_WRITETHROUGH**	
	CACHE_COPYBACK	(data cache only)
	CACHE_INH_PRECISE	(data cache only)
	CACHE_INH_IMPRECISE	(data cache only)
	CACHE_BURST_ENABLE	(data cache only)

The write-through, copy-back, serial, non-serial, precise and non precise modes change the state of the data transparent translation register (DTTR0) CM bits. Only DTTR0 is modified, since it typically maps DRAM space.

X86 PROCESSORS

The caching mode **CACHE_WRITETHROUGH** is available for the 486 family processors. The caching mode **CACHE_COPYBACK** becomes available for the P5(Pentium) family processors. The caching mode (**CACHE_COPYBACK** | **CACHE_SNOOP_ENABLE**) becomes available for the P6(PentiumPro, II, III) and P7(Pentium4) family processors.

POWER PC PROCESSORS

Modes should be set before caching is enabled. If two contradictory flags are set (for example, enable/disable), no action is taken for any of the input flags.

ARM PROCESSORS The caching capabilities and modes vary for members of the ARM processor family. All caches are provided on-chip, so cache support is mostly an architecture issue, not a BSP issue. However, the memory map is BSP-specific and some functions need knowledge of the memory map, so they have to be provided in the BSP.

ARM7TDMI (In ARM or Thumb state)
No cache or MMU at all. Dummy routine provided, so that **INCLUDE_CACHE_SUPPORT** can be defined (the default BSP configuration).

ARM710A
Combined instruction and data cache. Actually a write-through cache, but separate write-buffer effectively makes this a copy-back cache if the write-buffer is enabled. Use write-through/copy-back argument to decide whether to enable write buffer. Data and instruction cache modes must be identical.

ARM810
Combined instruction and data cache. Write-through and copy-back cache modes, but separate write-buffer effectively makes even write-through a copy-back cache as all writes are buffered, when cache is enabled. Data and instruction cache modes must be identical.

ARMSA110
Separate instruction and data caches. Write-through and copy-back cache mode for data, but separate write-buffer effectively makes even write-through a copy-back cache as all writes are buffered, when cache is enabled.

RETURNS **OK**

ERRNO Not Available

SEE ALSO **cacheArchLib**

cacheAuLibInit( )

NAME **cacheAuLibInit( )** – initialize the Au cache library

SYNOPSIS

```
STATUS cacheAuLibInit
    (
    CACHE_MODE instMode,        /* instruction cache mode */
    CACHE_MODE dataMode,        /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize
    )
```

DESCRIPTION

This routine initializes the function pointers for the Au cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheAuLibInit( )**.

RETURNS

OK.

ERRNO

Not Available

SEE ALSO

cacheAuLib

cacheClear( )

NAME

cacheClear( ) – clear all or some entries from a cache

SYNOPSIS

```
STATUS cacheClear
    (
    CACHE_TYPE cache,    /* cache to clear */
    void *     address,  /* virtual address */
    size_t     bytes     /* number of bytes to clear */
    )
```

DESCRIPTION

This routine flushes and invalidates all or some entries in the specified cache.

RETURNS

OK, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheDisable( )

NAME

cacheDisable( ) – disable the specified cache

SYNOPSIS

```
STATUS cacheDisable
    (
    CACHE_TYPE cache  /* cache to disable */
    )
```

DESCRIPTION

This routine flushes the cache and disables the instruction or data cache.

RETURNS **OK**, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO **S_cacheLib_INVALID_CACHE**
the cache type specified is invalid.

SEE ALSO **cacheLib**

cacheDmaFree( )

NAME **cacheDmaFree( )** – free the buffer acquired with **cacheDmaMalloc( )**

SYNOPSIS

```
STATUS cacheDmaFree
    (
    void * pBuf  /* pointer to malloc/free buffer */
    )
```

DESCRIPTION This routine frees the buffer returned by **cacheDmaMalloc( )**.

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheDmaMalloc( )

NAME **cacheDmaMalloc( )** – allocate a cache-safe buffer for DMA devices and drivers

SYNOPSIS

```
void * cacheDmaMalloc
    (
    size_t bytes  /* number of bytes to allocate */
    )
```

DESCRIPTION This routine returns a pointer to a section of memory that will not experience any cache coherency problems. Function pointers in the **CACHE_FUNCS** structure provide access to DMA support routines.

RETURNS A pointer to the cache-safe buffer, or **NULL**.

ERRNO N/A

SEE ALSO **cacheLib**

cacheDrvFlush( )

NAME **cacheDrvFlush( )** – flush the data cache for drivers

SYNOPSIS

```
STATUS cacheDrvFlush
    (
    CACHE_FUNCS * pFuncs,   /* pointer to CACHE_FUNCS */
    void *        address,  /* virtual address */
    size_t        bytes     /* number of bytes to flush */
    )
```

DESCRIPTION This routine flushes the data cache entries using the function pointer from the specified set.

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheDrvInvalidate( )

NAME **cacheDrvInvalidate( )** – invalidate data cache for drivers

SYNOPSIS

```
STATUS cacheDrvInvalidate
    (
    CACHE_FUNCS * pFuncs,   /* pointer to CACHE_FUNCS */
    void *        address,  /* virtual address */
    size_t        bytes     /* no. of bytes to invalidate */
    )
```

DESCRIPTION This routine invalidates the data cache entries using the function pointer from the specified set.

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheDrvPhysToVirt( )

NAME

cacheDrvPhysToVirt( ) – translate a physical address for drivers

SYNOPSIS

```
void * cacheDrvPhysToVirt
    (
    CACHE_FUNCS * pFuncs,  /* pointer to CACHE_FUNCS */
    void *        address  /* physical address */
    )
```

DESCRIPTION

This routine performs a physical-to-virtual address translation using the function pointer from the specified set.

RETURNS

The virtual address that maps to the physical address argument.

ERRNO

N/A

SEE ALSO

cacheLib

cacheDrvVirtToPhys( )

NAME

cacheDrvVirtToPhys( ) – translate a virtual address for drivers

SYNOPSIS

```
void * cacheDrvVirtToPhys
    (
    CACHE_FUNCS * pFuncs,  /* pointer to CACHE_FUNCS */
    void *        address  /* virtual address */
    )
```

DESCRIPTION

This routine performs a virtual-to-physical address translation using the function pointer from the specified set.

RETURNS

The physical address translation of a virtual address argument.

ERRNO

N/A

SEE ALSO

cacheLib

cacheEnable( )

NAME

cacheEnable( ) – enable the specified cache

SYNOPSIS

```
STATUS cacheEnable
    (
    CACHE_TYPE cache  /* cache to enable */
    )
```

DESCRIPTION

This routine invalidates the cache tags and enables the instruction or data cache.

RETURNS

OK, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheFlush( )

NAME

cacheFlush( ) – flush all or some of a specified cache

SYNOPSIS

```
STATUS cacheFlush
    (
    CACHE_TYPE cache,    /* cache to flush */
    void *     address,  /* virtual address */
    size_t     bytes     /* number of bytes to flush */
    )
```

DESCRIPTION

This routine flushes (writes to memory) all or some of the entries in the specified cache. Depending on the cache design, this operation may also invalidate the cache tags. For write-through caches, no work needs to be done since RAM already matches the cached entries. Note that write buffers on the chip may need to be flushed to complete the flush.

RETURNS

OK, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheForeignClear( )

NAME

cacheForeignClear( ) – clear foreign data from selected cache

SYNOPSIS

```
STATUS cacheForeignClear
    (
    CACHE_TYPE cache,     /* cache to clear */
    VIRT_ADDR  virtAddr,  /* virtual address */
    PHYS_ADDR  physAddr,  /* physical address */
    size_t     bytes      /* number of bytes to flush */
    )
```

DESCRIPTION

This routine performs a clear of the requested area of memory from the cache. Unlike **cacheClear( )**, this routine does not assume that the provided virtual address is valid within the current address space. The called routine may clear more data than is requested.

RETURNS

OK, or **ERROR** if the requested cache operation failed.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheForeignFlush( )

NAME

cacheForeignFlush( ) – flush foreign data from selected cache

SYNOPSIS

```
STATUS cacheForeignFlush
    (
    CACHE_TYPE cache,     /* cache to flush */
    VIRT_ADDR  virtAddr,  /* virtual address */
    PHYS_ADDR  physAddr,  /* physical address */
    size_t     bytes      /* number of bytes to flush */
    )
```

DESCRIPTION

This routine performs a flush of the requested area of memory from the cache. Unlike **cacheFlush( )**, this routine does not assume that the provided virtual address is valid within the current address space. This routine may flush more data than is requested, in order to ensure that the required data has been flushed from the cache.

RETURNS

OK, or **ERROR** if the requested cache operation failed.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheForeignInvalidate( )

NAME

cacheForeignInvalidate( ) – invalidate foreign data from selected cache

SYNOPSIS

```
STATUS cacheForeignInvalidate
    (
    CACHE_TYPE cache,     /* cache to flush */
    VIRT_ADDR  virtAddr,  /* virtual address */
    PHYS_ADDR  physAddr,  /* physical address */
    size_t     bytes      /* number of bytes to flush */
    )
```

DESCRIPTION

This routine performs an invalidate of the requested area of memory from the cache. Unlike **cacheInvalidate( )**, this routine does not assume that the provided virtual address is valid within the current address space. Unlike the flush and clear functions, it is a programming error to invalidate more data from the cache than is requested. For this reason, if the arhitecture does not provide its own foreign invalidation routine, this function emulates the operation using **cacheClear( )**.

RETURNS

OK, or **ERROR** if the requested cache operation failed.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

SEE ALSO

cacheLib

cacheInvalidate( )

NAME

cacheInvalidate( ) – invalidate all or some of a specified cache

SYNOPSIS

```
STATUS cacheInvalidate
    (
    CACHE_TYPE cache,    /* cache to invalidate */
    void *     address,  /* virtual address */
    size_t     bytes     /* number of bytes to invalidate */
    )
```

DESCRIPTION This routine invalidates all or some of the entries in the specified cache. Depending on the cache design, the invalidation may be similar to the flush, or one may invalidate the tags directly.

RETURNS **OK**, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO **S_cacheLib_INVALID_CACHE**
the cache type specified is invalid.

SEE ALSO **cacheLib**

cacheLibInit( )

NAME **cacheLibInit( )** – initialize the cache library for a processor architecture

SYNOPSIS

```
STATUS cacheLibInit
    (
    CACHE_MODE instMode,  /* inst cache mode */
    CACHE_MODE dataMode   /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the appropriate cache library. For architectures with more than one cache implementation, the board support package must select the appropriate cache library with **sysCacheLibInit**. Systems without cache coherency problems (i.e., bus snooping) should NULLify the flush and invalidate function pointers in the **cacheLib** structure to enhance driver and overall system performance. This can be done in **sysHwInit( )**.

RETURNS **OK**, or **ERROR** if there is no cache library installed.

ERRNO N/A

SEE ALSO **cacheLib**

cacheLock( )

NAME **cacheLock( )** – lock all or part of a specified cache

SYNOPSIS

```
STATUS cacheLock
```

```
    (
    CACHE_TYPE cache,    /* cache to lock */
    void *     address,  /* virtual address */
    size_t     bytes     /* number of bytes to lock */
    )
```

DESCRIPTION This routine locks all (global) or some (local) entries in the specified cache. Cache locking is useful in real-time systems. Not all caches can perform locking.

RETURNS **OK**, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO **S_cacheLib_INVALID_CACHE**
the cache type specified is invalid.

SEE ALSO **cacheLib**

cachePipeFlush( )

NAME **cachePipeFlush( )** – flush processor write buffers to memory

SYNOPSIS

```
STATUS cachePipeFlush (void)
```

DESCRIPTION This routine forces the processor output buffers to write their contents to RAM. A cache flush may have forced its data into the write buffers, then the buffers need to be flushed to RAM to maintain coherency.

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheR10kLibInit( )

NAME **cacheR10kLibInit( )** – initialize the R10000 cache library

SYNOPSIS

```
STATUS cacheR10kLibInit
    (
    CACHE_MODE instMode,        /* instruction cache mode */
    CACHE_MODE dataMode,        /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize,
    UINT32     sCacheSize,
    UINT32     sCacheLineSize
    )
```

DESCRIPTION This routine initializes the function pointers for the R10000 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheR10kLibInit( )**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **cacheR10kLib**

cacheR4kLibInit( )

NAME **cacheR4kLibInit( )** – initialize the R4000 cache library

SYNOPSIS

```
STATUS cacheR4kLibInit
    (
    CACHE_MODE instMode,        /* instruction cache mode */
    CACHE_MODE dataMode,        /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize,
    UINT32     sCacheSize,
    UINT32     sCacheLineSize
    )
```

DESCRIPTION This routine initializes the function pointers for the R4000 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheR4kLibInit( )**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **cacheR4kLib**

cacheR5kLibInit( )

NAME

cacheR5kLibInit( ) – initialize the R5000 cache library

SYNOPSIS

```
STATUS cacheR5kLibInit
    (
    CACHE_MODE instMode,         /* instruction cache mode */
    CACHE_MODE dataMode,         /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize,
    UINT32     sCacheSize,
    UINT32     sCacheLineSize
    )
```

DESCRIPTION

This routine initializes the function pointers for the R5000 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheR5kLibInit( )**.

RETURNS

OK.

ERRNO

Not Available

SEE ALSO

cacheR5kLib

cacheR7kLibInit( )

NAME

cacheR7kLibInit( ) – initialize the R7000 cache library

SYNOPSIS

```
STATUS cacheR7kLibInit
    (
    CACHE_MODE instMode,         /* instruction cache mode */
    CACHE_MODE dataMode,         /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize,
    UINT32     sCacheSize,
    UINT32     sCacheLineSize,
    UINT32     tCacheSize,
    UINT32     tCacheLineSize
    )
```

DESCRIPTION This routine initializes the function pointers for the R7000 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheR7kLibInit( )**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **cacheR7kLib**

cacheSh7750LibInit( )

NAME **cacheSh7750LibInit( )** – initialize the SH7750 cache library

SYNOPSIS

```
STATUS cacheSh7750LibInit
    (
    CACHE_MODE instMode,  /* instruction cache mode */
    CACHE_MODE dataMode   /* data cache mode */
    )
```

DESCRIPTION This routine initializes the cache library for the Renesas SH7750 processor. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled. If two complementary flags are set (enable/disable), no action is taken for any of the input flags.

The following caching modes are available for the SH7750, SH7750R and SH7770 processors:

SH7750 :	**CACHE_WRITETHROUGH**	
	CACHE_COPYBACK	(copy-back cache for P0/P3, data cache only)
	CACHE_COPYBACK_P1	(copy-back cache for P1, data cache only)
	CACHE_RAM_MODE	(use half of cache as RAM, data cache only)
	CACHE_A25_INDEX	(use A25 as MSB of cache index)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)
SH7750R:	**CACHE_WRITETHROUGH**	

	CACHE_COPYBACK	(copy-back cache for P0/P3, data cache only)
	CACHE_COPYBACK_P1	(copy-back cache for P1, data cache only)
	CACHE_RAM_MODE	(use half of cache as RAM, data cache only)
	CACHE_2WAY_MODE	(use RAM in 2way associ. mode, data cache only)
	CACHE_A25_INDEX	(use A25 as MSB of cache index)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)
SH7770 :	**CACHE_SH4A_MODE**	(SH4A cache support)
	CACHE_WRITETHROUGH	
	CACHE_COPYBACK	(copy-back cache for P0/P3, data cache only)
	CACHE_COPYBACK_P1	(copy-back cache for P1, data cache only)
	CACHE_2WAY_MODE	(use RAM in 2way associ. mode)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)

The CACHE_DMA_BYPASS_Px modes allow to allocate "cache-safe" buffers without MMU. If none of CACHE_DMA_BYPASS_Px modes is specified, **cacheDmaMalloc( )** returns a cache-safe buffer on logical space, which is created by the MMU. If **CACHE_DMA_BYPASS_P0** is selected, **cacheDmaMalloc( )** returns a cache-safe buffer on P2 space, and **cacheDmaFree( )** releases the buffer to P0 space. Namely, if the system memory partition is located on P0, cache-safe buffers can be allocated and freed without MMU, by selecting **CACHE_DMA_BYPASS_P0**.

RETURNS **OK**, or **ERROR** if specified cache mode is invalid.

ERRNO

SEE ALSO **cacheSh7750Lib**

cacheStoreBufDisable( )

NAME
cacheStoreBufDisable( ) – disable the store buffer (MC68060 only)

SYNOPSIS
```
void cacheStoreBufDisable (void)
```

DESCRIPTION
This routine resets the ESB bit of the Cache Control Register (CACR) to disable the store buffer.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
cacheArchLib

cacheStoreBufEnable( )

NAME
cacheStoreBufEnable( ) – enable the store buffer (MC68060 only)

SYNOPSIS
```
void cacheStoreBufEnable (void)
```

DESCRIPTION
This routine sets the ESB bit of the Cache Control Register (CACR) to enable the store buffer. To maximize performance, the four-entry first-in-first-out (FIFO) store buffer is used to defer pending writes to writethrough or cache-inhibited imprecise pages.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
cacheArchLib

cacheTextLocalUpdate( )

NAME
cacheTextLocalUpdate( ) – synchronize the caches on local cpu only

SYNOPSIS
```
STATUS cacheTextLocalUpdate
    (
    void * address,  /* virtual address */
    size_t bytes     /* number of bytes to sync */
    )
```

DESCRIPTION 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. The operation is limited to the local CPU (i.e., no CPC is performed).

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheTextUpdate( )

NAME **cacheTextUpdate( )** – synchronize the instruction and data caches

SYNOPSIS

```
STATUS cacheTextUpdate
    (
    void * address,  /* virtual address */
    size_t bytes     /* number of bytes to sync */
    )
```

DESCRIPTION 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.

RETURNS **OK**, or **ERROR** if the cache control is not supported.

ERRNO N/A

SEE ALSO **cacheLib**

cacheTx49LibInit( )

NAME **cacheTx49LibInit( )** – initialize the Tx49 cache library

SYNOPSIS

```
STATUS cacheTx49LibInit
    (
    CACHE_MODE instMode,        /* instruction cache mode */
    CACHE_MODE dataMode,        /* data cache mode */
    UINT32     iCacheSize,      /* instruction cache size */
    UINT32     iCacheLineSize,  /* instruction cache line size */
    UINT32     dCacheSize,      /* data cache size */
    UINT32     dCacheLineSize   /* data cache line size */
    )
```

DESCRIPTION This routine initializes the function pointers for the Tx49 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheTx49LibInit( )**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **cacheTx49Lib**

cacheUnlock( )

NAME **cacheUnlock( )** – unlock all or part of a specified cache

SYNOPSIS

```
STATUS cacheUnlock
    (
    CACHE_TYPE cache,    /* cache to unlock */
    void *     address,  /* virtual address */
    size_t     bytes     /* number of bytes to unlock */
    )
```

DESCRIPTION This routine unlocks all (global) or some (local) entries in the specified cache. Not all caches can perform unlocking.

RETURNS **OK**, or **ERROR** if the cache type is invalid or the cache control is not supported.

ERRNO **S_cacheLib_INVALID_CACHE**
the cache type specified is invalid.

SEE ALSO **cacheLib**

calloc( )

NAME **calloc( )** – allocate space for an array (ANSI)

SYNOPSIS

```
void * calloc
    (
    size_t elemNum,  /* number of elements */
    size_t elemSize  /* size of elements */
    )
```

DESCRIPTION This routine allocates a block of memory for an array that contains *elemNum* elements of size *elemSize*. This space is initialized to zeros.

RETURNS A pointer to the block, or **NULL** if the call fails.

ERRNO Possible errnos generated by this routine include:

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO **memPartLib**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: General Utilities* (**stdlib.h**)

cbioBlkCopy( )

NAME **cbioBlkCopy( )** – block to block (sector to sector) transfer routine

SYNOPSIS

```
STATUS cbioBlkCopy
    (
    CBIO_DEV_ID dev,        /* CBIO handle */
    block_t     srcBlock,   /* source start block */
    block_t     dstBlock,   /* destination start block */
    block_t     numBlocks   /* number of blocks to copy */
    )
```

DESCRIPTION This routine verifies the CBIO device is valid and if so calls the devices block to block transfer routine which makes copies of one or more blocks on the lower layer (hardware, subordinate CBIO, or **BLK_DEV**). It is optimized for block to block copies on the subordinate layer.

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** if successful or **ERROR** if the handle is invalid, or if the CBIO device routine returns **ERROR**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioBlkRW( )

NAME **cbioBlkRW( )** – transfer blocks to or from memory

SYNOPSIS

```
STATUS cbioBlkRW
    (
    CBIO_DEV_ID dev,          /* CBIO handle */
    block_t     startBlock,  /* starting block of transfer */
    block_t     numBlocks,   /* number of blocks to transfer */
    addr_t      buffer,      /* address of the memory buffer */
    CBIO_RW     rw,          /* direction of transfer R/W */
    cookie_t *  pCookie      /* pointer to cookie */
    )
```

DESCRIPTION This routine verifies the CBIO device is valid and if so calls the devices block transfer routine. The CBIO device performs block transfers between the device and memory.

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** if successful or **ERROR** if the handle is invalid, or if the CBIO device routine returns **ERROR**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioBytesRW( )

NAME **cbioBytesRW( )** – transfer bytes to or from memory

SYNOPSIS

```
STATUS cbioBytesRW
    (
    CBIO_DEV_ID dev,          /* CBIO handle */
    block_t     startBlock,  /* starting block of the transfer */
    off_t       offset,      /* offset into block in bytes */
    addr_t      buffer,      /* address of data buffer */
    size_t      nBytes,      /* number of bytes to transfer */
    CBIO_RW     rw,          /* direction of transfer R/W */
    cookie_t *  pCookie      /* pointer to cookie */
    )
```

DESCRIPTION This routine verifies the CBIO device is valid and if so calls the devices byte transfer routine which transfers between a user buffer and the lower layer (hardware, subordinate CBIO, or **BLK_DEV**). It is optimized for byte transfers.

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** if successful or **ERROR** if the handle is invalid, or if the CBIO device routine returns **ERROR**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioDevCreate( )

NAME **cbioDevCreate( )** – Initialize a CBIO device (Generic)

SYNOPSIS

```
CBIO_DEV_ID cbioDevCreate
    (
    caddr_t ramAddr,  /* where it is in memory (0 = KHEAP_ALLOC) */
    size_t  ramSize   /* pool size */
    )
```

DESCRIPTION This routine will create an empty **CBIO_DEV** structure and return a handle to that structure (**CBIO_DEV_ID**).

This routine is intended to be used by CBIO modules only. See **cbioLibP.h**

RETURNS **CBIO_DEV_ID** or **NULL** if **ERROR**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioDevVerify( )

NAME **cbioDevVerify( )** – verify **CBIO_DEV_ID**

SYNOPSIS

```
STATUS cbioDevVerify
    (
    CBIO_DEV_ID device  /* CBIO_DEV_ID to be verified */
    )
```

DESCRIPTION The purpose of this function is to determine if the device complies with the CBIO interface. It can be used to verify a CBIO handle before it is passed to **dosFsLib**, **rawFsLib**, **usrFdiskPartLib**, or other CBIO modules which expect a valid CBIO interface.

The device handle provided to this function, *device* is verified to be a CBIO device. If *device* is not a CBIO device **ERROR** is returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

The dcacheCbio and dpartCbio CBIO modules (and **dosFsLib**) use this function internally, and therefore this function need not be otherwise invoked when using compliant CBIO modules.

RETURNS **OK** or **ERROR** if not a CBIO device, if passed a **NULL** address, or if the check could cause an unaligned access.

ERRNO Not Available

SEE ALSO **cbioLib**, **dosFsLib**, **dcacheCbio( )**, **dpartCbio( )**

cbioIoctl( )

NAME **cbioIoctl( )** – perform ioctl operation on device

SYNOPSIS

```
STATUS cbioIoctl
    (
    CBIO_DEV_ID dev,      /* CBIO handle */
    int         command,  /* ioctl command to be issued */
    addr_t      arg       /* arg - specific to ioctl */
    )
```

DESCRIPTION This routine verifies the CBIO device is valid and if so calls the devices I/O control operation routine.

CBIO modules expect the following **ioctl( )** codes:

- **CBIO_RESET** - reset the CBIO device. When the third argument to the ioctl call accompanying **CBIO_RESET** is **NULL**, the code verifies that the disk is inserted and is ready, after getting it to a known state. When the 3rd argument is a non-zero, it is assumed to be a **BLK_DEV** pointer and **CBIO_RESET** will install a new subordinate block device. This work is performed at the **BLK_DEV** to CBIO layer, and all layers shall account for it. A **CBIO_RESET** indicates a possible change in device geometry, and the **CBIO_PARAMS** members will be reinitialized after a **CBIO_RESET**.
- **CBIO_STATUS_CHK** - check device status of CBIO device and lower layer
- **CBIO_DEVICE_LOCK** - Prevent disk removal
- **CBIO_DEVICE_UNLOCK** - Allow disk removal
- **CBIO_DEVICE_EJECT** - Unmount and eject device
- **CBIO_CACHE_FLUSH** - Flush any dirty cached data
- **CBIO_CACHE_INVAL** - Flush & Invalidate all cached data
- **CBIO_CACHE_NEWBLK** - Allocate scratch block

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** if successful or **ERROR** if the handle is invalid, or if the CBIO device routine returns **ERROR**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioLibInit( )

NAME **cbioLibInit( )** – Initialize CBIO Library

SYNOPSIS

```
STATUS cbioLibInit(void)
```

DESCRIPTION This function initializes the CBIO library, and will be called when the first CBIO device is created, hence it does not need to be called during system initialization. It can be called multiple times, but will do nothing after the first call.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **cbioLib**

cbioLock( )

NAME **cbioLock( )** – obtain CBIO device semaphore.

SYNOPSIS

```
STATUS cbioLock
    (
    CBIO_DEV_ID dev,     /* CBIO handle */
    int         timeout  /* timeout in ticks */
    )
```

DESCRIPTION If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** or **ERROR** if the CBIO handle is invalid or semTake fails.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioModeGet( )

NAME **cbioModeGet( )** – return the mode setting for CBIO device

SYNOPSIS

```
int cbioModeGet
    (
    CBIO_DEV_ID dev  /* CBIO handle */
    )
```

DESCRIPTION If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID** This routine is not protected by a semaphore.

This routine confirms if the current layer is a CBIO to BLKDEV wrapper or a CBIO to CBIO layer. Depending on the current layer it either returns the mode from **BLK_DEV** or calls **cbioModeGet( )** recursively.

RETURNS **O_RDONLY**, **O_WRONLY**, or **O_RDWR** or **ERROR**

ERRNO Not Available

SEE ALSO **cbioLib**

cbioModeSet( )

NAME **cbioModeSet( )** – set mode for CBIO device

SYNOPSIS

```
STATUS cbioModeSet
    (
    CBIO_DEV_ID dev,  /* CBIO handle */
    int         mode  /* O_RDONLY, O_WRONLY, or O_RDWR */
    )
```

DESCRIPTION Valid modes are **O_RDONLY**, **O_WRONLY**, or **O_RDWR**.

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID** This routine is not protected by a semaphore.

This routine confirms if the current layer is a CBIO to BLKDEV wrapper or a CBIO to CBIO layer. Depending on the current layer it either sets the mode of the **BLK_DEV** or calls **cbioModeSet( )** recursively.

RETURNS **OK** or **ERROR** if mode is not set.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioParamsGet( )

NAME **cbioParamsGet( )** – fill in **CBIO_PARAMS** structure with CBIO device parameters

SYNOPSIS

```
STATUS cbioParamsGet
    (
    CBIO_DEV_ID   dev,          /* CBIO handle */
    CBIO_PARAMS * pCbioParams  /* pointer to CBIO_PARAMS */
    )
```

DESCRIPTION If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** or **ERROR** if the CBIO handle is invalid.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioRdyChgdGet( )

NAME **cbioRdyChgdGet( )** – determine ready status of CBIO device

SYNOPSIS

```
int cbioRdyChgdGet
    (
    CBIO_DEV_ID dev  /* CBIO handle */
    )
```

DESCRIPTION For example

```
switch (cbioRdyChgdGet (cbioDeviceId))
    {
    case TRUE:
        printf ("Disk changed.\n");
        break;
    case FALSE:
        printf ("Disk has not changed.\n");
        break;
    case ERROR:
        printf ("Not a valid CBIO device.\n");
        break;
    default:
    break;
    }
```

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID** This routine is not protected by a semaphore.

This routine will check down to the driver layer to see if any lower layer has its ready changed bit set to **TRUE**. If so, this routine returns **TRUE**. If no lower layer has its ready changed bit set to **TRUE**, this layer returns **FALSE**.

RETURNS **TRUE** if device ready status has changed, else **FALSE** if the ready status has not changed, else **ERROR** if the **CBIO_DEV_ID** is invalid.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioRdyChgdSet( )

NAME **cbioRdyChgdSet( )** – force a change in ready status of CBIO device

SYNOPSIS

```
STATUS cbioRdyChgdSet
    (
    CBIO_DEV_ID dev,    /* CBIO handle */
    BOOL        status  /* TRUE/FALSE */
    )
```

DESCRIPTION Pass **TRUE** in status to force READY status change.

If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID** If status is not passed as **TRUE** or **FALSE**, **ERROR** is returned. This routine is not protected by a semaphore.

This routine sets readyChanged bit of passed **CBIO_DEV**.

RETURNS **OK** or **ERROR** if the device is invalid or status is not **TRUE** or **FALSE**.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioShow( )

NAME **cbioShow( )** – print information about a CBIO device

SYNOPSIS

```
STATUS cbioShow
    (
    CBIO_DEV_ID dev  /* CBIO handle */
    )
```

DESCRIPTION This function will display on standard output all information which is generic for all CBIO devices. See the CBIO modules particular device show routines for displaying implementation-specific information.

It takes two arguments:

A **CBIO_DEV_ID** which is the CBIO handle to display or **NULL** for the most recent device.

RETURNS **OK** or **ERROR** if no valid **CBIO_DEV** is found.

ERRNO Not Available

SEE ALSO **cbioLib**, **dcacheShow( )**, **dpartShow( )**

cbioUnlock( )

NAME **cbioUnlock( )** – release CBIO device semaphore.

SYNOPSIS

```
STATUS cbioUnlock
    (
    CBIO_DEV_ID dev  /* CBIO handle */
    )
```

DESCRIPTION If the **CBIO_DEV_ID** passed to this routine is not a valid CBIO handle, **ERROR** will be returned with errno set to **S_cbioLib_INVALID_CBIO_DEV_ID**

RETURNS **OK** or **ERROR** if the CBIO handle is invalid or the semGive fails.

ERRNO Not Available

SEE ALSO **cbioLib**

cbioWrapBlkDev( )

NAME **cbioWrapBlkDev( )** – create CBIO wrapper atop a **BLK_DEV** device

SYNOPSIS

```
CBIO_DEV_ID cbioWrapBlkDev
    (
    BLK_DEV * pDevice  /* BLK_DEV * device pointer */
    )
```

DESCRIPTION The purpose of this function is to make a blkIo (**BLK_DEV**) device comply with the CBIO interface via a wrapper.

The device handle provided to this function, *device* is verified to be a blkIo device. A lean CBIO to **BLK_DEV** wrapper is then created for a valid blkIo device. The returned **CBIO_DEV_ID** device handle may be used with **dosFsDevCreate( )**, **dcacheDevCreate( )**, and any other routine expecting a valid **CBIO_DEV_ID** handle.

To verify a blkIo pointer we see that all mandatory functions are not **NULL**.

Note that if a valid **CBIO_DEV_ID** is passed to this function, it will simply be returned without modification.

The **dosFsLib**, dcacheCbio, and dpartCbio CBIO modules use this function internally, and therefore this function need not be otherwise invoked when using those CBIO modules.

RETURNS a CBIO device pointer, or **NULL** if not a blkIo device

ERRNO Not Available

SEE ALSO **cbioLib**, **dosFsLib**, **dcacheCbio( )**, **dpartCbio( )**

cbrt( )

NAME **cbrt( )** – compute a cube root

SYNOPSIS

```
double cbrt
    (
    double x  /* value to compute the cube root of */
    )
```

DESCRIPTION This routine returns the cube root of *x* in double precision.

RETURNS The double-precision cube root of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

cbrtf( )

NAME **cbrtf( )** – compute a cube root

SYNOPSIS

```
float cbrtf
    (
    float x  /* argument */
    )
```

DESCRIPTION This routine returns the cube root of *x* in single precision.

RETURNS The single-precision cube root of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

cd( )

NAME **cd( )** – change the default directory

SYNOPSIS

```
STATUS cd
    (
    const char * name  /* new directory name */
    )
```

DESCRIPTION This command sets the default directory to *name*. The default directory is a device name, optionally followed by a directory local to that device.

NOTE This is a target resident function, which manipulates the target I/O system. It must be preceded with the @ letter if executed from the Host Shell (windsh), which has a built-in command of the same name that operates on the Host's I/O system.

To change to a different directory, specify one of the following:

- an entire path name with a device name, possibly followed by a directory name. The entire path name will be changed.
- a directory name starting with a **~** or **/** or **$**. The directory part of the path, immediately after the device name, will be replaced with the new directory name.
- a directory name to be appended to the current default directory. The directory name will be appended to the current default directory.

An instance of ".." indicates one level up in the directory tree.

Note that when accessing a remote file system via RSH or FTP, the VxWorks network device must already have been created using **netDevCreate( )**.

WARNING The **cd( )** command does very little checking that *name* represents a valid path. If the path is invalid, **cd( )** may return **OK**, but subsequent calls that depend on the default path will fail.

EXAMPLES The following example changes the directory to device **/fd0/**:

```
-> cd "/fd0/"
```

This example changes the directory to device **wrs:** with the local directory **~leslie/target**:

```
-> cd "wrs:~leslie/target"
```

After the previous command, the following changes the directory to **wrs:~leslie/target/config**:

```
-> cd "config"
```

After the previous command, the following changes the directory to **wrs:~leslie/target/demo**:

```
-> cd "../demo"
```

After the previous command, the following changes the directory to **wrs:/etc**.

```
-> cd "/etc"
```

Note that **~** can be used only on network devices (RSH or FTP).

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrFsLib**, **pwd( )**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

cdromFsDevCreate( )

NAME **cdromFsDevCreate( )** – create a CD-ROM filesystem (cdromFs) I/O device.

SYNOPSIS

```
CDROM_VOL_DESC_ID cdromFsDevCreate
    (
    char *   devName,  /* device name */
    device_t device    /* underlying block device handle */
    )
```

DESCRIPTION This routine creates an instance of a cdromFs device in the I/O system. As input, this function requires an eXtended Block Device (XBD) identifier (device_t) for the CD drive on which to create a cdromFs I/O device. Thus, **xxxXbdDevCreate( )**, for example, should have already been called to create the XBD device. Alternatively, **xxxBlkDevCreate( )**, for example, can be called to create a legacy **BLK_DEV** driver followed by **xbdBlkDevCreate( )** to create an XBD wrapper around the **BLK_DEV** device.

RETURNS **CDROM_VOL_DESC_ID**, or **NULL** if error.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**

SEE ALSO **cdromFsLib**, **cdromFsInit( )**

cdromFsDevDelete( )

NAME

cdromFsDevDelete( ) – delete a CD-ROM filesystem (cdromFs) I/O device

SYNOPSIS

```
STATUS cdromFsDevDelete
    (
    CDROM_VOL_DESC_ID pVolDesc  /* ptr to CDROM_VOL_DESC */
    )
```

DESCRIPTION

This routine deletes the specified volume. This involves removing the "device" from the I/O system, and freeing all resources associated with the volume.

RETURNS

OK if specified volume was successfully deleted, otherwise **ERROR**

ERRNO

Not Available

SEE ALSO

cdromFsLib, **cdromFsInit( )**, **cdromFsDevCreate( )**

cdromFsInit( )

NAME

cdromFsInit( ) – initialize the VxWorks CD-ROM file system

SYNOPSIS

```
STATUS cdromFsInit
    (
    UINT32 commonBufferSize  /* common buffer size */
    )
```

DESCRIPTION

This routine initializes the VxWorks CD-ROM file system. It is automatically called when the **INCLUDE_CDROMFS** component is configured into the system.

RETURNS

OK or **ERROR**, if driver can not be installed.

ERRNO

S_iosLib_DRIVER_GLUT

SEE ALSO

cdromFsLib, **cdromFsDevCreate( )**, **iosLib.h**

cdromFsVersionDisplay( )

NAME **cdromFsVersionDisplay( )** – display the cdromFs version number

SYNOPSIS

```
void cdromFsVersionDisplay
    (
    int level  /* level of display, not used */
    )
```

DESCRIPTION This routine displays the cdromFs version number. This routine has been deprecated.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cdromFsLib**, **cdromFsVersionNumGet( )**, **cdromFsVolConfigShow( )**

cdromFsVersionNumGet( )

NAME **cdromFsVersionNumGet( )** – return the cdromFs version number

SYNOPSIS

```
uint32_t cdromFsVersionNumGet
    (
    void
    )
```

DESCRIPTION This routine returns the cdromFs version number. This routine has been deprecated.

RETURNS the cdromFs version number.

ERRNO Not Available

SEE ALSO **cdromFsLib**, **cdromFsVersionDisplay( )**

cdromFsVolConfigShow( )

NAME **cdromFsVolConfigShow( )** – show the volume configuration information

SYNOPSIS

```
VOID cdromFsVolConfigShow
```

```
    (
    void * arg  /* device name or CDROM_VOL_DESC * */
    )
```

DESCRIPTION This routine retrieves the volume configuration for the named **cdromFsLib** device and prints it to standard output. The information displayed is retrieved from the **BLK_DEV** structure for the specified device.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cdromFsLib**, N/A

ceilf( )

NAME **ceilf( )** – compute the smallest integer greater than or equal to a specified value (ANSI)

SYNOPSIS

```
float ceilf
    (
    float v  /* value to find the ceiling of */
    )
```

DESCRIPTION This routine returns the smallest integer greater than or equal to *v*, in single precision.

RETURNS The smallest integral value greater than or equal to *v*, in single precision.

ERRNO Not Available

SEE ALSO **mathALib**

cfree( )

NAME **cfree( )** – free a block of memory from the system memory partition (kernel heap)

SYNOPSIS

```
STATUS cfree
    (
    char * pBlock  /* pointer to block of memory to free */
    )
```

DESCRIPTION

This routine returns to the free memory pool a block of memory previously allocated with **calloc( )**.

It is an error to free a memory block that was not previously allocated.

RETURNS

OK, or **ERROR** if the the block is invalid.

ERRNO

Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

SEE ALSO

memPartLib

checkStack( )

NAME

checkStack( ) – print a summary of each task's stack usage

SYNOPSIS

```
void checkStack
    (
    int taskNameOrId  /* task name or task ID; 0 = summarize all */
    )
```

DESCRIPTION

This command displays a summary of stack usage for a specified task, or for all tasks if no argument is given. The summary includes the total stack size (SIZE), the current number of stack bytes used (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used at the top of the stack (MARGIN = SIZE - HIGH).

Both the execution and the exception stack information are displayed. The exception stack is used by the task when it gets an exception or by a process task when it enters a system call and executes kernel code.

For example:

```
-> checkStack tShell0
  NAME        ENTRY        TID       SIZE   CUR  HIGH  MARGIN
------------ ------------ ---------- ----- ----- ----- ------
tShell0      shellTask    0x60351ba8 77824  6272 14144  63680
(Exception Stack)                    12096     0   680  11416
value = 1614093224 = 0x60351ba8
```

The maximum stack usage is determined by scanning down from the top of the stack for the first byte whose value is not 0xee. In VxWorks, when a task is spawned, all bytes of a task's stack are initialized to 0xee. The task's stack will not be filled with 0xee if the task option **VX_NO_STACK_FILL** is specified or if the kernel configuration parameter **VX_GLOBAL_NO_STACK_FILL** is set to **TRUE**.

DEFICIENCIES
It is possible for a task to write beyond the end of its stack, but not write into the last part of its stack. This will not be detected by **checkStack( )**.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
usrLib, **taskSpawn( )**, the VxWorks programmer guides.

chkdsk( )

NAME
chkdsk( ) – perform consistency checking on a MS-DOS file system

SYNOPSIS
```
STATUS chkdsk
    (
    const char * pDevName,     /* device name */
    u_int        repairLevel,  /* how to fix errors */
    u_int        verbose       /* verbosity level */
    )
```

DESCRIPTION
This function invokes the integral consistency checking built into the **dosFsLib** file system, via FIOCHKDSK ioctl. During the test, the volume will be un-mounted and re-mounted, invalidating file descriptors to prevent any application code from accessing the volume during the test. If the drive was exported, it will need to be re-exported again as its file descriptors were also invalidated. Furthermore, the test will emit messages describing any inconsistencies found on the disk, as well as some statistics, depending upon the value of the *verbose* argument. Depending upon the value of *repairLevel*, the inconsistencies will be repaired, and changes written to disk.

These are the values for *repairLevel*:

0
Same as **DOS_CHK_ONLY** (1)

DOS_CHK_ONLY (1)
Only report errors, do not modify disk.

DOS_CHK_REPAIR (2)
Repair any errors found.

These are the values for *verbose*:

0
similar to **DOS_CHK_VERB_1**

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 as errors encountered during the test.

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 that the consistency check procedure will *unmount* the file system, meaning the all currently open file descriptors will be deemed unusable.

RETURNS
OK or **ERROR** if device can not be checked or could not be repaired.

ERRNO
Not Available

SEE ALSO
usrFsLib, **dosFsLib**, the VxWorks programmer guides.

chmod( )

NAME
chmod( ) – change the permission mode of a file

SYNOPSIS
```
int chmod
    (
    const char *path,  /* path of the file */
    mode_t     mode    /* mode bits to change */
    )
```

DESCRIPTION
The chmod utility changes or assigns the mode of a file. The mode of a file specifies its permissions and other attributes. Note that this routine receives **path of the file whose mode needs to be changed** as the first argument compairing to fchmod routine.

The value of *mode* is bitwise inclusive OR of the permissions to be assigned

These permission constants are defined in *sys/stat.h* as follows:

S_IRUSR
Read permission, owner.

S_IWUSR
Write permission, owner.

S_IXUSR
Execute/search permission, owner.

S_IRWXU
Read/write/execute permission, owner.

S_IRGRP
Read permission, group.

S_IWGRP
Write permission, group.

S_IXGRP
Execute/search permission, group.

S_IRWXG
Read/write/execute permission, group.

S_IROTH
Read permission, other.

S_IWOTH
Write permission, other.

S_IXOTH
Execute/search permission, other.

S_IRWXO
Read/write/execute permission, other.

RETURNS

If it succeeds, returns **OK**, 0. Otherwise, **ERROR**, -1 is returned, errno is set to indicate the error and no change is done to the file.

The following example changes the mode of the file "myFile" to owner Read/write/execute, group Read and other Read:

```
status = chmod ("myFile", S_IRWXU | S_IRGRP | S_IROTH );
```

ERRNO

ENOENT
Either *path* is an empty string or **NULL** pointer.

ELOOP
Circular symbolic link of *path*, or too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in *path*.

others
Other errors reported by device driver of *path*.

SEE ALSO

fsPxLib

clock_getres( )

NAME **clock_getres( )** – get the clock resolution (POSIX)

SYNOPSIS

```
int clock_getres
    (
    clockid_t         clock_id,  /* clock ID */
    struct timespec * res        /* where to store resolution */
    )
```

DESCRIPTION This routine gets the clock resolution, in nanoseconds, based on the rate returned by **sysClkRateGet( )**. If *res* is non-**NULL**, the resolution is stored in the location pointed to.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *clock_id* is invalid.

ERRNO **EINVAL**

SEE ALSO **clockLib**, **clock_settime( )**, **sysClkRateGet( )**, **clock_setres( )**

clock_gettime( )

NAME **clock_gettime( )** – get the current time of the clock (POSIX)

SYNOPSIS

```
int clock_gettime
    (
    clockid_t         clock_id,  /* clock ID */
    struct timespec * tp         /* where to store current time */
    )
```

DESCRIPTION This routine gets the current value *tp* for the clock.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *clock_id* is invalid or *tp* is **NULL**.

ERRNO **EINVAL**
EFAULT

SEE ALSO **clockLib**

clock_nanosleep( )

NAME

clock_nanosleep( ) – high resolution sleep with specifiable clock

SYNOPSIS

```
int clock_nanosleep
    (
    clockid_t                clock_id,
    int                      flags,
    const struct timespec *  rqtp,
    struct timespec *        rmtp
    )
```

DESCRIPTION

If the flag **TIMER_ABSTIME** is not set in *flags*, this function causes the current thread to be delayed until either the time interval specified by *rqtp* has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal handler, or the process is terminated. The clock used to measure the time is the clock specified by *clock_id*.

If the flag **TIMER_ABSTIME** is set in *flags*, this function causes the current thread to be delayed until either the time value of the clock specified by *clock_id* reaches the absolute time specified by *rqtp*, or a signal is delivered to the calling thread whose action is to invoke a signal handler, or the process is terminated. If at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of *clock_id*, this function returns immediately without delaying the calling process.

The delay caused by this function may be longer than requested because *rqtp* is rounded up to an integer multiple of the timer resolution, or because of the scheduling of other tasks by the system. Except for the case of being interrupted by a signal, the suspension time for the relative delay (i.e. if **TIMER_ABSTIME** is not set) is not less than the time interval *rqtp*, as measured by the corresponding clock.

If a signal is caught by the calling task while sleeping for a relative time delay (i.e. flag **TIMER_ABSTIME** is not set in the *flags* argument), and the *rmtp* argument is non-**NULL**, the timespec structure referenced by *rmtp* is updated to contain the amount of time remaining in the interval. This is the requested sleep time minus the time actually slept.

This function only supports **CLOCK_REALTIME** and **CLOCK_MONOTONIC** clocks.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL
tp is outside the supported range, or the *tp* nanosecond value is less than 0 or equal to or greater than 1,000,000,000.

EINTR
The sleep was interrupted by receiving a signal .

ENOTSUP
The *clock_id* value is not supported.

SEE ALSO **clockLib**, **clock_getres( )**

clock_setres( )

NAME **clock_setres( )** – set the clock resolution

SYNOPSIS

```
int clock_setres
    (
    clockid_t         clock_id,  /* clock ID */
    struct timespec * res        /* resolution to be set */
    )
```

DESCRIPTION This routine is obsolete. It will always return **OK**.

NOTE Non-POSIX.

RETURNS **OK** always.

ERRNO **EINVAL**

SEE ALSO **clockLib**, **clock_getres( )**, **sysClkRateSet( )**

clock_settime( )

NAME **clock_settime( )** – set the clock to a specified time (POSIX)

SYNOPSIS

```
int clock_settime
    (
    clockid_t               clock_id,  /* clock ID */
    const struct timespec * tp         /* time to set */
    )
```

DESCRIPTION This routine sets the clock to the value *tp*, which should be a multiple of the clock resolution. If *tp* is not a multiple of the resolution, it is truncated to the next smallest multiple of the resolution.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *clock_id* is invalid, *tp* is outside the supported range, or the *tp* nanosecond value is less than 0 or equal to or greater than 1,000,000,000.

ERRNO **EINVAL**

SEE ALSO **clockLib**, **clock_getres( )**

close( )

NAME **close( )** – close a file

SYNOPSIS

```
STATUS close
    (
    int fd  /* file descriptor to close */
    )
```

DESCRIPTION This routine closes the specified file and frees the file descriptor. It calls the device driver to do the work.

RETURNS The status of the driver close routine, or **ERROR** if the file descriptor is invalid.

ERRNO

EBADF
Invalid file descriptor.

Others
Other errors generated by device drivers.

SEE ALSO **ioLib**

closedir( )

NAME **closedir( )** – close a directory (POSIX)

SYNOPSIS

```
STATUS closedir
    (
    DIR *pDir  /* pointer to directory descriptor */
    )
```

DESCRIPTION This routine closes a directory which was previously opened using **opendir( )**. The *pDir* parameter is the directory descriptor pointer that was returned by **opendir( )**.

RETURNS **OK** or **ERROR**, the result of the **close( )** command.

ERRNO

EBADF
Invalid file descriptor.

Others
Other errors generated by device drivers.

SEE ALSO **dirLib**, **opendir( )**, **readdir( )**, **rewinddir( )**

cnsAppRegister( )

NAME **cnsAppRegister( )** – Registers an application with the CNS library.

SYNOPSIS
```
STATUS cnsAppRegister
    (
    char *                        pName,
    CNS_APP_READ_FUNCPTR          readFunc,
    CNS_APP_WRITE_FUNCPTR         writeFunc,
    CNS_MEDIATYPE_ADD_FUNCPTR mediaTypeAddFunc,
    CNS_MEDIATYPE_REMOVE_FUNCPTR mediaTypeRmFunc,
    CNS_DATA_PARSE_FUNCPTR        dataParse
    )
```

DESCRIPTION This routine registers an application that uses CNS services. The information to be passed during registration includes the application name and function pointers to access the application's local objects.

ARGUMENTS **pName** identifies the application.

readFunc
writeFunc
haveServer - Specifies whether the application implements its own read server.

mediaTypeAddFunc - Reuired only if **haveServer** is set to **TRUE**. It points to a function to notify the application of the addition of a new media type.

mediaTypeRmFunc - Reuired only if **haveServer** is set to **TRUE**. It points to a function to notify the application of the removal of a media type.

RETURNS **OK** or **ERROR** if no more media can be added or **pAppInfo** is **NULL** or one or more of the fields of **cnsAppInfo_t** are **NULL**.

ERRNO ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsLib**

cnsClose( )

NAME **cnsClose( )** – Close or create and open named communication medium for read/write.

SYNOPSIS

```
STATUS cnsClose
    (
    cnsMediaId_t * pMediaId
    )
```

DESCRIPTION This routine closes the named communication medium.

ARGUMENTS **pMediaId** is a pointer to an instance of the **cnsMediaId_t**, which describes the media being closed. **pMediaId->connId** cannot be 0.

RETURNS **OK** or **ERROR** if the named media cannot be closed.

ERRNO **cnsClose** maintains the ERRNO of the underlying media for media close errors. ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsLib**

cnsCompLibInit( )

NAME **cnsCompLibInit( )** – Initialize the CNS COMP library.

SYNOPSIS

```
STATUS cnsCompLibInit
    (
    void
    )
```

DESCRIPTION This routine initializes the CNS COMP library.

ARGUMENTS N/A

RETURNS **OK** or **ERROR** if media registration fails.

ERRNO ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsCompLib**

cnsDefaultMediaTypeSet( )

NAME **cnsDefaultMediaTypeSet( )** – Set the default media type.

SYNOPSIS

```
STATUS cnsDefaultMediaTypeSet
    (
    char * pName
    )
```

DESCRIPTION This routine sets the default media type to that specified by **pName**. The default media type is used in cases where channel access routines are called without specifying the media type.

ARGUMENTS **pName** specifies the media type to which the default is set.

Returns:
ERROR if the specified media type does not exist or **OK**.

RETURNS Not Available

ERRNO **S_cnsLib_MEDIATYPE_INVALID** if the specified media type does not exist.
S_cnsLib_GEN_ERROR

SEE ALSO **cnsLib**

cnsLibInit( )

NAME **cnsLibInit( )** – Initialize the CNS library.

SYNOPSIS

```
STATUS cnsLibInit
    (
    ulong_t maxMediaCount
    )
```

DESCRIPTION This routine initializes the CNS library.

ARGUMENTS **maxMediaCount**, if non-0, specifies the maximum number of media types to support. If 0, **cnsLib** uses the default **CNS_MAX_MEDIA_TYPES**, which is currently defined in '**cnsCfg.h**.

RETURNS **OK** or **ERROR** if write error occurs.

ERRNO ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsLib**

cnsMediaRegister( )

NAME **cnsMediaRegister( )** – Registers a communication media with the CNS.

SYNOPSIS

```
STATUS cnsMediaRegister
    (
    cnsMediaInfo_t * pMediaInfo
    )
```

DESCRIPTION This routine registers a communication media with the CNS. The information to be passed during registration includes the medium type name and function pointers that are used to access the communication medium.

ARGUMENTS **pMediaInfo** is a pointer to an instance of the **cnsMediaInfo_t**, which specifies the medium name and the pointers to medium access functions.

RETURNS **OK** or **ERROR** if no more media can be added or **pMediaInfo** is **NULL** or one or more of the fields of **cnsMediaInfo_t** are **NULL**.

ERRNO ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsLib**

cnsMediaTypeRemove( )

NAME **cnsMediaTypeRemove( )** – Remove a media type from an application's media list.

DESCRIPTION **cnsMediaRegister( )** calls this routine everytime a new media type is removed to the media list.

ARGUMENTS **pApp**, points to an instance of the **cnsApp_t** structure, which describes the application.

type identifies the media type being added.

RETURNS **OK** or **ERROR**.

ERRNO ERRNO for CNS-internal errors are TBD.

TODO: Must implement this.

```
STATUS cnsMediaTypeRemove
   (
   cnsApp_t *  pApp,
   lont    type
   )
{ }
```

SEE ALSO **cnsLib**

cnsMediumTypeNext( )

NAME

cnsMediumTypeNext( ) – Return the name of the media type next in the list.

SYNOPSIS

```
char * cnsMediumTypeNext
    (
    char * pName
    )
```

DESCRIPTION

This routine returns the name pf the media type next to that specified by **pName**. If **pName** is **NULL** or if **pName** points to an empty string, the name of the first media type in the list is returned.

ARGUMENTS

pName specifies the name of the media type whose successor in the list is being sought.

RETURNS

NULL if the media type specified by **pName** is the last in the list;
the name of the media type next to that specified by **pName** otherwise.

ERRNO

Not Available

SEE ALSO

cnsLib

cnsMsgEncode( )

NAME

cnsMsgEncode( ) – Encode a message as understood by CNS.

SYNOPSIS

```
STATUS cnsMsgEncode
    (
    long      type,       /* CNS_MSGTYPE_READREQ */
    char *    pData,
    char *    pFormat,
    ulong_t * pFormatLen
    )
```

DESCRIPTION

This routine formats a message ad understood by CNS. A CNS message starts with the message type and is followed by optional data field which is specific to the message data.

There are five message types: read request messafe, write request message, write with ack request message, reply **OK** message, and reply **ERROR** message.

The request messages (read, write, and wite with ack) are typically sent by clients. The reply messages are sent by notification processor applications. A read request message is always followed by a reply message. The reply message starts with a status string (ok or error) and is followed by associated data.

ARGUMENTS

type specifies the message data type. The value of **type** can be one of following:

- o **CNS_REQ_READ** - specifies a read request.
- o **CNS_REQ_WRITE** - specifies a write request.
- o **CNS_REQ_WRITE_WACK** - specifies write request with ack back.
- o **CNS_REPLY_OK** - specifes a reply to a successful read request or an ack to a successful write request.
- o **CNS_REPLY_ERR** - specifies a failed read request or a failed write request.

pData points to the data associated with the message type.

For read requests, the data consists typically of information of what is to be read. For example, for CSM, the data consists of the component's name and the object to be read.

For write requests, the data consists typically of information of what is to be modified and the modification data.

pformat points to a buffer to contain the formatted data.

pFormatLen specifies the capacity of **pFormat** during input, and specifies the actual length of the format when the **cnsMsgEncode( )** returns.

create, if **TRUE**, specifies that the medium is to be created.

RETURNS

OK or **ERROR** if the named media cannot be opened/created.

ERRNO

cnsMsgEncode maintains the ERRNO of the underlying media for media open errors. ERRNO for CNS-internal errors are TBD.

SEE ALSO

cnsLib

cnsOpen( )

NAME

cnsOpen( ) – Open or create and open named communication medium for read/write.

SYNOPSIS

```
STATUS cnsOpen
    (
    cnsMediaId_t * pMediaId,
    BOOL           create,
    long *         pConnState
    )
```

DESCRIPTION

Depending upon the value of **create**, this routine opens or creates and then opens the named communication medium for read/write.

ARGUMENTS

pMedia is a pointer to an instance of the **cnsMediaId_t**, which describes the media being created/opened. If open/create of the media succeeds, the **mediaId** and **connId** fields are updated to identify the open medium type and connection ID.

create, if **TRUE**, specifies that the medium is to be created.

RETURNS

OK or **ERROR** if the named media cannot be opened/created.

ERRNO

cnsOpen maintains the ERRNO of the underlying media for media open errors. ERRNO for CNS-internal errors are TBD.

SEE ALSO

cnsLib

cnsRead( )

NAME

cnsRead( ) – Read from a communication medium.

SYNOPSIS

```
STATUS cnsRead
    (
    cnsMediaId_t * pMediaId,
    char *         pReqInfo,
    char *         pBuf,
    ulong_t        bufLen,
    ulong_t *      pReadBytes
    )
```

DESCRIPTION This routine reads from a communication medium.

ARGUMENTS **pMediaId** is a pointer to an instance of the **cnsMediaId_t**, which describes the media being read from. **connId** of **pMediaId** cannot be 0. If **pName** is provided, then **cnsRead** verifies that the name matches the corresponding **mediaId** and **connId**. If they dont match, **ERROR** is returned.

pBuf points to a buffer to which the read data is to be copied.

bufLen specifies the capacity of **pBuf** in bytes.

pReadBytes points to a buffer to which the number of actually read bytes is to be copied.

RETURNS **OK** or **ERROR** if error occured while reading, or if **pMediaId->connId** is 0 or if **pMediaId->pName** is not **NULL** and it is not consistent with the corresponding **pMediaId->connId** and **pMediaId->mediaId**.

ERRNO **cnsRead** maintains the ERRNO of the underlying media for media read errors. ERRNO for CNS-internal errors are TBD.

SEE ALSO **cnsLib**

cnsWrite( )

NAME **cnsWrite( )** – Write to a communication medium.

SYNOPSIS

```
STATUS cnsWrite
    (
    cnsMediaId_t * pMediaId,
    char *         pReqInfo,
    char *         pDataBuf,
    ulong_t        bufLen,
    char *         pReplyBuf,
    ulong_t *      pReplyLen
    )
```

DESCRIPTION This routine writes to a communication medium.

ARGUMENTS

pMediaId is a pointer to an instance of the **cnsMediaId_t**, which describes the media being written to. **connId** of **pMediaId** cannot be 0. If **pName** is provided, then **cnsWrite** verifies that the name matches the corresponding **mediaId** and **connId**. If they dont match, **ERROR** is returned.

pBuf points to a buffer from which the data to be written is to be copied.

bufLen specifies the capacity of **pBuf** in bytes.

pReplyLen if greater than 0, it implies that the caller expects an acknowledgement back. The message type is then set to **CNS_REQ_WRITE_WACK**.

RETURNS

OK or **ERROR** if error occured while reading, or if **pMediaId->connId** is 0 or if **pMediaId->pName** is not **NULL** and it is not consistent with the corresponding **pMediaId->connId** and **pMediaId->mediaId**.

ERRNO

cnsWrite maintains the ERRNO of the underlying media for media write errors. ERRNO for CNS-internal errors are TBD.

SEE ALSO

cnsLib

commit( )

NAME

commit( ) – commit current transaction to disk.

SYNOPSIS

```
STATUS commit
    (
    const char * pDevName  /* name of the device to commit */
    )
```

DESCRIPTION

This command is for transactional based file systems only such as HRFS. It is a shortcut for the ioctl function FIOCOMMITFS which commits the current transaction to disk to make changes permanment.

EXAMPLE

```
    -> commit "/ata0a"             /* commit transaction on "/fd0" */
```

RETURNS

OK, or **ERROR** if the device is not formatted with a file system that does not support the FIOCOMMITFS ioctl function or *pDevName* is not valid.

ERRNO

Not Available

SEE ALSO

usrFsLib, **hrFsLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

---

copy( )

NAME

copy( ) – copy *in* (or stdin) to *out* (or stdout)

SYNOPSIS

```
STATUS copy
    (
    const char * in,  /* name of file to read  (if NULL assume stdin)  */
    const char * out  /* name of file to write (if NULL assume stdout) */
    )
```

DESCRIPTION

This command copies from the input file to the output file, until an end-of-file is reached.

EXAMPLES

The following example displays the file **dog**, found on the default file device:

```
-> copy <dog
```

This example copies from the console to the file **dog**, on device **/ct0/**, until an **EOF** (default ^D) is typed:

```
-> copy >/ct0/dog
```

This example copies the file **dog**, found on the default file device, to device **/ct0/**:

```
-> copy <dog >/ct0/dog
```

This example makes a conventional copy from the file named **file1** to the file named **file2**:

```
-> copy "file1", "file2"
```

Remember that standard input and output are global; therefore, spawning the first three constructs will not work as expected.

RETURNS

OK, or **ERROR** if *in* or *out* cannot be opened/created, or if there is an error copying from *in* to *out*.

ERRNO

Not Available

SEE ALSO

usrFsLib, **copyStreams( )**, **tyEOFSet( )**, **cp( )**, **xcopy( )**, the VxWorks programmer guides.

copyStreams( )

NAME **copyStreams( )** – copy from/to specified streams

SYNOPSIS

```
STATUS copyStreams
    (
    int inFd,  /* file descriptor of stream to copy from */
    int outFd  /* file descriptor of stream to copy to */
    )
```

DESCRIPTION This command copies from the stream identified by *inFd* to the stream identified by *outFd* until an end of file is reached in *inFd*. This command is used by **copy( )**.

RETURNS **OK**, or **ERROR** if there is an error reading from *inFd* or writing to *outFd*.

ERRNO Not Available

SEE ALSO **usrFsLib**, **copy( )**, the VxWorks programmer guides.

coreDumpClose( )

NAME **coreDumpClose( )** – close a core dump

SYNOPSIS

```
STATUS coreDumpClose
    (
    CORE_DUMP_ID coreDumpId  /* ID returned by coreDumpOpen() */
    )
```

DESCRIPTION This routine frees resources allocated by **coreDumpOpen( )**.

RETURNS **OK** or **ERROR** if *coreDumpId* is invalid

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpCopy( )

NAME **coreDumpCopy( )** – copy a core dump to the given path

SYNOPSIS
```
STATUS coreDumpCopy
    (
    UINT32 coreDumpIndex,  /* core dump index, or 0 for all */
    char * destPath        /* destination path */
    )
```

DESCRIPTION This routine copies the core dump specified by *coreDumpIndex* to *destPath*. If *coreDumpIndex* is 0, then all the core dumps available on device are copied. If *destPath* is **NULL**, then the destination path is current directory.

coreDumpIndex is the index of the core dump returned by **coreDumpNextGet( )** routine.

RETURNS **OK**, or **ERROR** if a core dump index is invalid, or if the copy failed.

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpCreateHookAdd( )

NAME **coreDumpCreateHookAdd( )** – add a routine to be called at every core dump create

SYNOPSIS
```
STATUS coreDumpCreateHookAdd
    (
    FUNCPTR createHook  /* routine to be called when a core dump is created
*/
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called whenever a core dump is created. Upon creation, all routines specified by **coreDumpCreateHookAdd( )** will be called.

The routine should be declared as follows:

```
    STATUS createHook (void)
```

RETURNS **OK**, or **ERROR** if the table of core dump create routines is full.

ERRNO **S_coreDumpLib_CORE_DUMP_HOOK_TABLE_FULL**
core dump create hook table is full

SEE ALSO **coreDumpHookLib**, **coreDumpCreateHookDelete( )**

coreDumpCreateHookDelete( )

NAME **coreDumpCreateHookDelete( )** – delete a previously added core dump create routine

SYNOPSIS

```
STATUS coreDumpCreateHookDelete
    (
    FUNCPTR createHook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each core dump create.

RETURNS **OK**, or **ERROR** if the routine is not in the table of core dump create routines.

ERRNO **S_coreDumpLib_CORE_DUMP_HOOK_NOT_FOUND**
core dump create hook can not be found

SEE ALSO **coreDumpHookLib**, **coreDumpCreateHookAdd( )**

coreDumpDevFormat( )

NAME **coreDumpDevFormat( )** – format the core dump device

SYNOPSIS

```
STATUS coreDumpDevFormat
    (
    UINT32 coreDumpMax  /* Maximum number of core dump on device */
    )
```

DESCRIPTION This routine formats the core dump device. Formatting a core dump device consists of erasing all the core dumps available on device. If an **erase( )** routine is specified for the core dump device, then this routine will be called to erase the whole device (useful for flash devices for example); otherwise the only part of the core dump storage will be erased using the underlying device write command.

Once the core dump device has been erased, **coreDumpDevFormat( )** reformat it to support the given number of core dump.

NOTE It is not possible to erase only one core dump from the device, the full device is erased.

RETURNS **OK**, or **ERROR** if the format operation failed.

ERRNO N/A

SEE ALSO **coreDumpLib**

coreDumpDevShow( )

NAME **coreDumpDevShow( )** – display information on core dump device

SYNOPSIS

```
STATUS coreDumpDevShow (void)
```

DESCRIPTION This routine displays basic information on the core dump device. It displays the current number of core dumps stored on the device, the maximum number of core dumps that can be stored on the device. The total size of the core dump device and the free size on this device.

RETURNS **OK**, or **ERROR**.

ERRNO N/A

SEE ALSO **coreDumpShow**

coreDumpInfoGet( )

NAME **coreDumpInfoGet( )** – get information on a core dump

SYNOPSIS

```
CORE_DUMP_INFO * coreDumpInfoGet
    (
    UINT32 coreDumpIndex  /* core dump index */
    )
```

DESCRIPTION This routine retrieves information on a given core dump. It allocates a **CORE_DUMP_INFO** structure and fills it with the retrieved information, a pointer to this structure is then returned, and it will be up to the caller to free this structure when the information will have been consumed.

coreDumpIndex is the index of the core dump returned by **coreDumpNextGet( )** routine.

```
typedef struct core_dump_info            /* core dump information */
    {
```

```
    UINT32                 coreDumpIndex;/* core dump index */
    BOOL                   valid;        /* core dump validity */
    UINT32                 errnoVal;     /* core dump errno */
    char                   name[MAX_CORE_DUMP_LEN];
                                         /* name of the core dump */
    size_t                 size;         /* size of the core dump */
    CORE_DUMP_TYPE         type;         /* origin of the core dump */
    int                    taskId;       /* task Id (kernel core dump) */
    char                   taskName[MAX_CORE_DUMP_TASK_LEN];
                                         /* name of task */
    UINT32                 rtpId;        /* process Id (process core  dump)
*/
    char                   rtpName[MAX_CORE_DUMP_RTP_LEN];
                                         /* path to RTP */
    int                    excNum;       /* exception number (Not valid for
*/
                                         /* on-demand &  Panic core dumps) */
    UINT32                 pc;           /* exception program counter */
    UINT32                 sp;           /* exception stack pointer */
    UINT32                 fp;           /* exception frame pointer */
    time_t                 time;         /* generation calendar time */
    UINT32                 ticks;        /* VxWorks time stamp */
    CORE_DUMP_CKSUM_STATUS cksumStatus;  /* core dump checksum status */
    char             infoString[MAX_CORE_DUMP_INFO_LEN];
                                   /* information string */
    BOOL             excIsValid;   /* Indicate validity of exception */
                                   /* information (excNum, pc, sp, fp) */
    } CORE_DUMP_INFO;
```

The *valid* bit in **CORE_DUMP_INFO** structure indicates if the core dump was successfully written on the storage or if there was an error writing it; if a core dump has been only partially written (because the storage is too small for example) then it will be marked as invalid and the *size* field will represent the size that has been actually written on the storage. If the core dump is marked as not valid, the *errnoVal* field contains the errno that has been set while generating the core dump.

The *type* field indicates the origin of the core dump. Here are the possible values:

```
typedef enum                      /* Core Dump Type */
    {
    CORE_DUMP_USER,               /* 0: user coredump (on-demand) */
    CORE_DUMP_KERNEL_INIT,        /* 1: fatal error during kernel intialization
*/
    CORE_DUMP_INTERRUPT,          /* 2: Obsolete / Kept for backward compat */
    CORE_DUMP_KERNEL_PANIC,       /* 3: Obsolete / Kept for backward compat */
    CORE_DUMP_KERNEL_TASK,        /* 4: kernel task error */
    CORE_DUMP_RTP,                /* 5: process coredump */
    CORE_DUMP_KERNEL              /* 6: VxWorks kernel error */
    } CORE_DUMP_TYPE;
```

The *cksumStatus* field indicates the status of the core dump checksum. Here are the possible values:

```
typedef enum                      /* Core Dump Checksum Status */
    {
    CORE_DUMP_NO_CKSUM,           /* 0: No cksum available (N/A) */
    CORE_DUMP_CKSUM_OK,         /* 1: Core dump checksum status OK */
    CORE_DUMP_CKSUM_ERROR       /* 2: Core dump checksum status ERROR */
    } CORE_DUMP_CKSUM_STATUS;
```

If the core dump checksum status is equal to **CORE_DUMP_NO_CKSUM**, then this means that the checksum facility was not enabled on the target (**CORE_DUMP_CKSUM_ENABLE** parameter of **INCLUDE_CORE_DUMP** component).

RETURNS A pointer to a **CORE_DUMP_INFO** structure, or **NULL** if failed to read core dump information from the core dump device.

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpIsAvailable( )

NAME **coreDumpIsAvailable( )** – is a core dump available for retrieval

SYNOPSIS

```
BOOL coreDumpIsAvailable (void)
```

DESCRIPTION This routine can be called to determine if at least one core dump is available on core dump device for retrieval.

RETURNS **TRUE** if there is at least one core dump available for retrieval or **FALSE** if there is no core dump available on device.

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpMemDump( )

NAME **coreDumpMemDump( )** – dump an area of memory in VxWorks core dump

SYNOPSIS

```
STATUS coreDumpMemDump
    (
    void * buffer,  /* address of memory region to dump */
    size_t size,    /* size of memory region to dump */
    void * vaddr    /* destination address in core dump */
    )
```

DESCRIPTION This routine can be called by a user to dump an additional area of memory in a core dump. The area of memory to dump is specified by *buffer* and *size* arguments. The *vaddr* allows to specify at which address in the core dump, the area of memory must be mapped.

This routine must be called from a VxWorks core dump creation hook.

No verification is done to check that an area of memory is already available in a core dump, an area of memory can be dumped several times in the core dump.

If a core dump memory filter has been installed and filter the provided memory area or part of it, the filtered area will not be written in core dump.

RETURNS **OK** or **ERROR** if core dump generation has failed

ERRNOS **S_coreDumpLib_CORE_DUMP_GENERATE_NOT_RUNNING**
This routine is not called from a core dump creation hook

SEE ALSO **coreDumpLib**

coreDumpMemFilterAdd( )

NAME **coreDumpMemFilterAdd( )** – add a memory region filter

SYNOPSIS

```
STATUS coreDumpMemFilterAdd
    (
    void * addr,  /* address of memory region to filter */
    size_t size   /* size of memory region to filter */
    )
```

DESCRIPTION This routine adds a filter to exclude a memory region from core dump.

RETURNS **OK**, or **ERROR** if the maximum number of memory region filter is reached

ERRNO **S_coreDumpLib_CORE_DUMP_FILTER_TABLE_FULL**
Core dump memory filter table is full

SEE ALSO **coreDumpMemFilterLib**

coreDumpMemFilterDelete( )

NAME

coreDumpMemFilterDelete( ) – delete a memory region filter

SYNOPSIS

```
STATUS coreDumpMemFilterDelete
    (
    void * addr,  /* address of memory region to filter */
    size_t size   /* size of memory region to filter */
    )
```

DESCRIPTION

This routine deletes the core dump memory region filter specified by the *addr* and *size* arguments.

RETURNS

OK, or **ERROR** if the specified filter does not exist.

ERRNO

S_coreDumpLib_CORE_DUMP_FILTER_NOT_FOUND
Filter not found in core dump filter table

SEE ALSO

coreDumpMemFilterLib

coreDumpNextGet( )

NAME

coreDumpNextGet( ) – get the next core dump on device

SYNOPSIS

```
STATUS coreDumpNextGet
    (
    UINT32   currentIdx,  /* current core dump index or 0 to */
                          /* get first core dump */
    UINT32 * pNextIdx     /* where to store next core dump */
                          /* index */
    )
```

DESCRIPTION

This routine retrieves the index of the next core dump available on the device; this routine can be used to walk through the core dump list. The first call of this routine must be performed with *currentIdx* equal to zero to get the index of the first core dump found on the device; the returned index is stored at the memory location pointed to by *pNextIdx*. The returned index can then be used by other routines like **coreDumpOpen( )**, **coreDumpInfoGet( )** or by **coreDumpNextGet( )** to get the the index of the following core dump. If there is no next core dump, the returned index is set to 0.

RETURNS

OK or **ERROR** if there was an error reading the core dump list.

ERRNO

N/A

SEE ALSO **coreDumpUtilLib**

coreDumpOpen( )

NAME **coreDumpOpen( )** – open an existing core dump for retrieval

SYNOPSIS

```
CORE_DUMP_ID coreDumpOpen
    (
    UINT32 coreDumpIndex  /* core dump index */
    )
```

DESCRIPTION This routine opens an existing core dump. The core dump content can then be retrieved using **coreDumpRead( )**. The core dump must then be closed using **coreDumpClose( )**. Some memory is allocated dynamically by **coreDumpOpen( )** routine, it will be freed by **coreDumpClose( )** routine.

coreDumpIndex is the index of the core dump returned by **coreDumpNextGet( )** routine.

RETURNS A core dump ID, or **NULL** if open has failed.

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpRead( )

NAME **coreDumpRead( )** – read from a core file

SYNOPSIS

```
int coreDumpRead
    (
    CORE_DUMP_ID coreDumpId,  /* ID returned by coreDumpOpen() */
    void *       buffer,      /* where to store read data */
    size_t       size         /* number of bytes to read */
    )
```

DESCRIPTION This routine reads a number of bytes (less than or equal to *size*) from the specified core dump ID and places them in *buffer*.

RETURNS The number of bytes read (between 1 and *size*, 0 if end of file), or **ERROR** if the core dump ID does not exist, of if there was an error reading the core dump.

ERRNO N/A

SEE ALSO **coreDumpUtilLib**

coreDumpShow( )

NAME **coreDumpShow( )** – display information on core dumps

SYNOPSIS

```
STATUS coreDumpShow
    (
    UINT32 coreDumpIndex,  /* core dump index */
    UINT32 level           /* core dump show level */
    )
```

DESCRIPTION This routine displays basic information on the given core dump. If *coreDumpIndex* is 0, then it displays basic information on all core dumps available on the core dump device. If *level* is 1, then detailled information are displayed.

The core dump display contains the following fields:

Field	Meaning
NAME	Core dump name on device
IDX	Core dump index for retrieval
VALID	Indicate if core dump is valid or not
ERRNO	If core dump is not valid, errno set during core dump generation
SIZE	Size of the core dump (compressed size if compression is enabled)
CKSUM	Core dump checksum status
TYPE	Type of the core dump
TASK	Name of the task at the origin of the core dump

The following table describes the different core dump types:

Field	Meaning
USER	User coredump
KERNEL_INIT	Fatal error during kernel intialization
KERNEL_TASK	Kernel task error
RTP	Process coredump
KERNEL	VxWorks kernel error
UNKNOWN	Unknown coredump type

The following table describes the different core dump checksum status:

Field	Meaning
N/A	No cksum available

Field	Meaning
OK	Core dump checksum status **OK**
ERROR	Core dump checksum status **ERROR**

EXAMPLE

```
-> coreDumpShow 0,1
NAME        IDX VALID   ERRNO      SIZE     CKSUM    TYPE        TASK
----------- --- ----- ---------- ---------- ----- ------------ ----------
vxcore1.z     1 Y            N/A 0x000ef05b OK    KERNEL_TASK  t1

Core Dump detailled information:
--------------------------------
Time:                   THU JAN 01 00:01:42 1970 (ticks = 6174)
Task:                   "t1" (0x611e0a20)
Process:                "(Kernel)" (0x6017a500)
Description:            fatal kernel task-level exception!
Exception number:       0xb
Program counter:        0x6003823e
Stack pointer:          0x604d3da8
Frame pointer:          0x604d3fb0

value = 0 = 0x0
```

RETURNS **OK**, or **ERROR** if the core dump device is not initialized, not formatted, if **coreDumpShow( )** failed to retrieve information on a core dump, or if the *coreDumpIndex* is not a valid core dump index.

ERRNO N/A

SEE ALSO **coreDumpShow**

coreDumpUsrGenerate( )

NAME **coreDumpUsrGenerate( )** – generate a user (on-demand) core dump

SYNOPSIS

```
STATUS coreDumpUsrGenerate (void)
```

DESCRIPTION This routine generates a user (on-demand) core dump. When this routine is called, the core dump will be generated, stored on the configured storage and the target will be rebooted.

RETURNS **OK** or **ERROR** if core dump generation failed.

ERRNO N/A

SEE ALSO **coreDumpLib**

cosf( )

NAME **cosf( )** – compute a cosine (ANSI)

SYNOPSIS

```
float cosf
    (
    float x  /* angle in radians */
    )
```

DESCRIPTION This routine returns the cosine of *x* in single precision. The angle *x* is expressed in radians.

RETURNS The single-precision cosine of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

coshf( )

NAME **coshf( )** – compute a hyperbolic cosine (ANSI)

SYNOPSIS

```
float coshf
    (
    float x  /* value to compute the hyperbolic cosine of */
    )
```

DESCRIPTION This routine returns the hyperbolic cosine of *x* in single precision.

RETURNS The single-precision hyperbolic cosine of *x* if the parameter is greater than 1.0, or NaN if the parameter is less than 1.0.

Special cases:
If *x* is +INF, -INF, or NaN, **coshf( )** returns *x*.

ERRNO Not Available

SEE ALSO **mathALib**

cp( )

NAME **cp( )** – copy file into other file/directory.

SYNOPSIS

```
STATUS cp
    (
    const char * src,  /* source file or wildcard pattern */
    const char * dest  /* destination file name or directory */
    )
```

DESCRIPTION This command copies from the input file to the output file. If destination name is directory, a source file is copied into this directory, using the last element of the source file name to be the name of the destination file.

This function is very similar to **copy( )**, except it is somewhat more similar to the UNIX "cp" program in its handling of the destination.

src may contain a wildcard pattern, in which case all files matching the pattern will be copied to the directory specified in *dest*. This function does not copy directories, and is not recursive. To copy entire subdirectories recursively, use **xcopy( )**.

EXAMPLES

```
-> cp( "/sd0/FILE1.DAT","/sd0/dir2/f001.dat")
-> cp( "/sd0/dir1/file88","/sd0/dir2")
-> cp( "/sd0/*.tmp","/sd0/junkdir")
```

RETURNS **OK** or **ERROR** if destination is not a directory while *src* is a wildcard pattern, or if any of the files could not be copied.

ERRNO Not Available

SEE ALSO **usrFsLib**, **xcopy( )**, the VxWorks programmer guides.

cplusCallNewHandler( )

NAME **cplusCallNewHandler( )** – call the allocation failure handler (C++)

SYNOPSIS

```
extern void cplusCallNewHandler (void)
```

DESCRIPTION This function provides a procedural-interface to the new-handler. It can be used by user-defined new operators to call the current new-handler. This function is specific to VxWorks and may not be available in other C++ environments.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

cplusCtors( )

NAME **cplusCtors( )** – call static constructors (C++)

SYNOPSIS

```
extern "C" void cplusCtors
    (
    const char * moduleName  /* name of loaded module */
    )
```

DESCRIPTION This function is used to call static constructors under the manual strategy (see **cplusXtorSet( )**). *moduleName* is the name of an object module that was "munched" before loading. If *moduleName* is 0, then all static constructors, in all modules loaded by the VxWorks module loader, are called.

EXAMPLES The following example shows how to initialize the static objects in modules called "applx.out" and "apply.out".

```
    -> cplusCtors "applx.out"
    value = 0 = 0x0
    -> cplusCtors "apply.out"
    value = 0 = 0x0
```

The following example shows how to initialize all the static objects that are currently loaded, with a single invocation of **cplusCtors( )**:

```
    -> cplusCtors
    value = 0 = 0x0
```

WARNING **cplusCtors( )** should only be called once per module otherwise unpredictable behavior may result.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**, **cplusXtorSet( )**

cplusCtorsLink( )

NAME **cplusCtorsLink( )** – call all linked static constructors (C++)

SYNOPSIS

```
extern "C" void cplusCtorsLink (void)
```

DESCRIPTION This function calls constructors for all of the static objects linked with a VxWorks bootable image. When creating bootable applications, this function should be called from **usrRoot( )** to initialize all static objects. Correct operation depends on correctly munching the C++ modules that are linked with VxWorks.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

cplusDemanglerSet( )

NAME **cplusDemanglerSet( )** – change C++ demangling mode (C++)

SYNOPSIS

```
extern "C" void cplusDemanglerSet
    (
    int mode
    )
```

DESCRIPTION This command sets the C++ demangling mode to *mode*. The default mode is 2.

There are three demangling modes, *complete*, *terse*, and *off*. These modes are represented by numeric codes:

Mode	Code
off	0
terse	1
complete	2

In complete mode, when C++ function names are printed, the class name (if any) is prefixed and the function's parameter type list is appended.

In terse mode, only the function name is printed. The class name and parameter type list are omitted.

In off mode, the function name is not demangled.

EXAMPLES The following example shows how one function name would be printed under each demangling mode:

Mode	Printed symbol
off	_member__5classFPFl_PvPFPv_v
terse	_member
complete	foo::_member(void* (*)(long),void (*)(void*))

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

cplusDemanglerStyleSet( )

NAME **cplusDemanglerStyleSet( )** – change C++ demangling style (C++)

SYNOPSIS

```
extern "C" void cplusDemanglerStyleSet
    (
    DEMANGLER_STYLE style
    )
```

DESCRIPTION This command sets the C++ demangling mode to *style*. The available demangler styles are enumerated in **demangler.h**. The default demangling style depends on the toolchain used to build the kernel. For example if the Diab toolchain is used to build the kernel then the default demangler style is **DMGL_STYLE_DIAB**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

cplusDtors( )

NAME **cplusDtors( )** – call static destructors (C++)

SYNOPSIS

```
extern "C" void cplusDtors
```

```
(
const char * moduleName
)
```

DESCRIPTION This function is used to call static destructors under the manual strategy (see **cplusXtorSet( )**). *moduleName* is the name of an object module that was "munched" before loading. If *moduleName* is 0, then all static destructors, in all modules loaded by the VxWorks module loader, are called.

EXAMPLES The following example shows how to destroy the static objects in modules called "applx.out" and "apply.out":

```
-> cplusDtors "applx.out"
value = 0 = 0x0
-> cplusDtors "apply.out"
value = 0 = 0x0
```

The following example shows how to destroy all the static objects that are currently loaded, with a single invocation of **cplusDtors( )**:

```
-> cplusDtors
value = 0 = 0x0
```

WARNING **cplusDtors( )** should only be called once per module otherwise unpredictable behavior may result.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**, **cplusXtorSet( )**

cplusDtorsLink( )

NAME **cplusDtorsLink( )** – call all linked static destructors (C++)

SYNOPSIS

```
extern "C" void cplusDtorsLink (void)
```

DESCRIPTION This function calls destructors for all of the static objects linked with a VxWorks bootable image. When creating bootable applications, this function should be called during system shutdown to decommission all static objects. Correct operation depends on correctly munching the C++ modules that are linked with VxWorks.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

cplusLibInit( )

NAME **cplusLibInit( )** – initialize the C++ library (C++)

SYNOPSIS

```
extern "C" STATUS cplusLibInit (void)
```

DESCRIPTION This routine initializes the C++ library and forces all C++ run-time support to be linked with the bootable VxWorks image. If the configuration macro **INCLUDE_CPLUS** is defined, **cplusLibInit( )** is called automatically from the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **cplusLib**

cplusXtorGet( )

NAME **cplusXtorGet( )** – get the c++ Xtors strategy

SYNOPSIS

```
extern "C" int cplusXtorGet (void)
```

DESCRIPTION This function can be used to retrieve the current value of the C++ Xtors strategy.

RETURNS 1 for automatic or 0 for manual

ERRNO Not Available

SEE ALSO **cplusLib**, **cplusXtorSet( )**

cplusXtorSet( )

NAME

cplusXtorSet( ) – change C++ static constructor calling strategy (C++)

SYNOPSIS

```
extern "C" void cplusXtorSet
    (
    int strategy
    )
```

DESCRIPTION

This command sets the C++ static constructor calling strategy to *strategy*. The default strategy is 1.

There are two static constructor calling strategies: *automatic* and *manual*. These modes are represented by numeric codes:

Strategy	Code
manual	0
automatic	1

Under the manual strategy, a module's static constructors and destructors are called by **cplusCtors( )** and **cplusDtors( )**, which are themselves invoked manually.

Under the automatic strategy, a module's static constructors are called as a side-effect of loading the module using the VxWorks module loader. A module's static destructors are called as a side-effect of unloading the module.

NOTE

The manual strategy is applicable only to modules that are loaded by the VxWorks module loader. Static constructors and destructors contained by modules linked with the VxWorks image are called using **cplusCtorsLink( )** and **cplusDtorsLink( )**.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

cplusLib, **cplusXtorGet( )**

cpsr( )

NAME

cpsr( ) – return the contents of the current processor status register (ARM)

SYNOPSIS

```
int cpsr
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the status register from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

RETURNS The contents of the current processor status register.

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

cpuPwrMgrEnable( )

NAME **cpuPwrMgrEnable( )** – Set the CPU light power management to ON/OFF

SYNOPSIS

```
STATUS cpuPwrMgrEnable
    (
    BOOL enable
    )
```

DESCRIPTION This routine enables or disables the light CPU power manager based on the *enable* argument that is passed. When *enable* is **TRUE** the power manager is enabled. When *enable* is **FALSE** the power manager is disable. When it is enabled the power manager puts the CPU in the C1 state (non-executing state) when the VxWorks kernel becomes idle, that is, when there are no ISRs to process or tasks to dispatch. The "C1 state" is a term borrowed from the Advance Configuration and Power Interface (ACPI) specification. When in a non-executing state the CPU reduces its power consumption. The light power manager is automatically enabled when the VxWorks kernel boots. Therefore this routine need not be called unless a kernel application wishes to disable the power manager or to re-enable it after having disabled it. The only reason that could explain this routine returning **ERROR** is if the light power manager's initialization routine failed, which more than likely indicates that another power manager is configured in the VxWorks kernel. There can only be one power manager in a system.

RETURNS **OK** or **ERROR** if the power manager could not be enabled.

ERRNO N/A

SEE ALSO **cpuPwrLightLib**, **cpuPwrMgrIsEnabled( )**

cpuPwrMgrIsEnabled( )

NAME

cpuPwrMgrIsEnabled( ) – Get the CPU power management status

SYNOPSIS

```
BOOL cpuPwrMgrIsEnabled (void)
```

DESCRIPTION

This routine returns the status of the light power manager. **TRUE** is returned when it is enabled. Otherwise **FALSE** is returned.

RETURNS

TRUE or **FALSE**

ERRNO

N/A

SEE ALSO

cpuPwrLightLib, **cpuPwrMgrEnable( )**

creat( )

NAME

creat( ) – create a file

SYNOPSIS

```
int creat
    (
    const char *name,  /* name of the file to create */
    int        flag    /* file permissions */
    )
```

DESCRIPTION

This routine creates a file called *name* and opens it with a specified *flag*. This routine determines on which device to create the file; it then calls the create routine of the device driver to do most of the work. Therefore, much of what transpires is device/driver-dependent.

The parameter *flag* is set to **O_RDONLY** (0), **O_WRONLY** (1), **O_RDWR** (2) for the duration of time the file is open.

The parameter *flag* can be set to **O_SYNC**, on dosFs volumes, indicating that each write should be immediately written to the backing media. This flag synchronizes the FAT and the directory entries.

On NFS and POSIX compliant file systems such as HRFS, the parameter *flag* refers instead to the UNIX style file permission bits.

NOTE

For more information about situations when there are no file descriptors available, see the reference entry for **iosInit( )**.

RETURNS

A file descriptor number, or **ERROR** if a filename is not specified, the device does not exist, no file descriptors are available, or the driver returns **ERROR**.

ERRNO

ELOOP
Circular symbolic link, too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in path.

others
Other errors reported by device drivers.

SEE ALSO

ioLib, **open( )**

cret( )

NAME

cret( ) – continue until the current subroutine returns

SYNOPSIS

```
STATUS cret
    (
    int taskNameOrId  /* task to continue, 0 = default */
    )
```

DESCRIPTION

This routine places a breakpoint at the return address of the current subroutine of a specified task, then continues execution of that task.

To execute, enter:

```
    -> cret [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

When the breakpoint is hit, information about the task will be printed in the same format as in single-stepping. The breakpoint is automatically removed when hit, or if the task hits another breakpoint first.

RETURNS

OK, or **ERROR** if there is no such task or the breakpoint table is full.

ERRNO

N/A

SEE ALSO

dbgLib, **so( )**, **c( )**, **b( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

d( )

NAME **d( )** – display memory

SYNOPSIS

```
void d
    (
    void * adrs,    /* address to display (if 0, display next block */
    int    nunits,  /* number of units to print (if 0, use default) */
    int    width    /* width of displaying unit (1, 2, 4, 8) */
    )
```

DESCRIPTION This command displays the contents of memory, starting at *adrs*. If *adrs* is omitted or zero, **d( )** displays the next memory block, starting from where the last **d( )** command completed.

Memory is displayed in units specified by *width*. If *nunits* is omitted or zero, the number of units displayed defaults to last use. If *nunits* is non-zero, that number of units is displayed and that number then becomes the default. If *width* is omitted or zero, it defaults to the previous value. If *width* is an invalid number, it is set to 1. The valid values for *width* are 1, 2, 4, and 8. The number of units **d( )** displays is rounded up to the nearest number of full lines.

RETURNS N/A

ERRNO N/A

ERRNO N/A

SEE ALSO **usrLib**, **m( )**, the VxWorks programmer guides.

d0( )

NAME **d0( )** – return the contents of register **d0** (also **d1** - **d7**) (MC680x0)

SYNOPSIS

```
int d0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of register **d0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all data registers (**d0** - **d7**): **d0( )** - **d7( )**.

RETURNS The contents of register **d0** (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, \the VxWorks programmer guides

dbgBpTypeBind( )

NAME **dbgBpTypeBind( )** – bind a breakpoint handler to a breakpoint type (MIPS R3000, R4000, R4650)

SYNOPSIS

```
STATUS dbgBpTypeBind
    (
    int     bpType,  /* breakpoint type */
    FUNCPTR routine  /* function to bind */
    )
```

DESCRIPTION Dynamically bind a breakpoint handler to breakpoints of type 0 - 7. By default only breakpoints of type zero are handled with the vxWorks breakpoint handler (see **dbgLib**). Other types may be used for Ada stack overflow or other such functions. The installed handler must take the same parameters as **excExcHandle( )** (see **excLib**).

RETURNS **OK**, or **ERROR** if *bpType* is out of bounds.

ERRNO Not Available

SEE ALSO **dbgArchLib**, **dbgLib**, **excLib**

dbgHelp( )

NAME **dbgHelp( )** – display debugging help menu

SYNOPSIS

```
void dbgHelp (void)
```

DESCRIPTION This routine displays a summary of **dbgLib** utilities with a short description of each, similar to the following:

```
    dbgHelp                             Print this list
    dbgInit                             Install debug facilities
    b                                   Display breakpoints
    b         addr[,task[,count]]       Set breakpoint
```

```
e         addr[,eventNo[,task[,func[,arg]]]]] Set eventpoint (WindView)
bd        addr[,task]            Delete breakpoint
bdall     [task]                 Delete all breakpoints
c         [task[,addr[,addr1]]] Continue from breakpoint
cret      [task]                 Continue to subroutine return
s         [task[,addr[,addr1]]] Single step
so        [task]                 Single step/step over subroutine
l         [adr[,nInst]]          List disassembled memory
tt        [task]                 Do stack trace on task
bh        addr[,access[,task[,count[,quiet]]]]] set hardware breakpoint
                                 (if supported by the architecture)
```

RETURNS N/A

ERRNO N/A

SEE ALSO **dbgLib**, *VxWorks Kernel Programmer's Guide: Kernel Shell*

dbgInit( )

NAME **dbgInit( )** – initialize the shell debugging package

SYNOPSIS

```
STATUS dbgInit (void)
```

DESCRIPTION This routine initializes the shell debugging package and enables the basic breakpoint and single-step functions.

This routine also enables the shell abort function.

NOTE The debugging package should be initialized before any debugging routines are used. If the configuration macro **INCLUDE_DEBUG** is defined, **dbgInit( )** is called by the VxWorks root task at initialisation time.

RETURNS **OK**, or **ERROR** if the debug facility cannot be initialized.

ERRNO N/A

SEE ALSO **dbgLib**, *VxWorks Kernel Programmer's Guide: Kernel Shell*

dcacheDevCreate( )

NAME **dcacheDevCreate( )** – Create a disk cache

SYNOPSIS

```
CBIO_DEV_ID dcacheDevCreate
    (
    CBIO_DEV_ID subDev,    /* block device handle */
    char  *     pRamAddr,  /* where it is in memory (NULL = KHEAP_ALLOC)  */
    int         memSize,   /* amount of memory to use */
    char *      pDesc      /* device description string */
    )
```

DESCRIPTION This routine creates a CBIO layer disk data cache instance. The disk cache unit accesses the disk through the subordinate CBIO device driver, provided with the *subDev* argument.

A valid block device **BLK_DEV** handle may be provided instead of a CBIO handle, in which case it will be automatically converted into a CBIO device by using the wrapper functionality from **cbioLib**.

Memory which will be used for caching disk data may be provided by the caller with *pRamAddr*, or it will be allocated by **dcacheDevCreate( )** from the common system memory pool, if *memAddr* is passed as **NULL**. *memSize* is the amount of memory to use for disk caching, if 0 is passed, then a certain default value will be calculated, based on available memory. *pDesc* is a string describing the device, used later by **dcacheShow( )**, and is useful when there are many cached disk devices.

A maximum of 16 disk cache devices are supported at this time.

RETURNS disk cache device handle, or **NULL** if there is not enough memory to satisfy the request, or the *blkDev* handle is invalid.

ERRNO Not Available

SEE ALSO **dcacheCbio**

dcacheDevDisable( )

NAME **dcacheDevDisable( )** – Disable the disk cache for this device

SYNOPSIS

```
STATUS dcacheDevDisable
    (
    CBIO_DEV_ID dev  /* CBIO device handle */
    )
```

DESCRIPTION This function disables the cache by setting the bypass count to zero and storing the old value, if there is already an old value then we won't repeat the process though.

RETURNS **OK** if cache is sucessfully disabled or **ERROR**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dcacheCbio**

dcacheDevEnable( )

NAME **dcacheDevEnable( )** – Reenable the disk cache

SYNOPSIS

```
STATUS dcacheDevEnable
    (
    CBIO_DEV_ID dev  /* CBIO device handle */
    )
```

DESCRIPTION This function re-enables the cache if we disabled it. If we did not disable it, then we cannot re-enable it.

RETURNS **OK** if cache is sucessfully enabled or **ERROR**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dcacheCbio**

dcacheDevMemResize( )

NAME **dcacheDevMemResize( )** – set a new size to a disk cache device

SYNOPSIS

```
STATUS dcacheDevMemResize
    (
    CBIO_DEV_ID dev,     /* device handle */
    size_t      newSize  /* new cache size in bytes */
    )
```

DESCRIPTION

This routine is used to resize the dcache layer. This routine is also useful after a disk change event, for example a PCMCIA disk swap. The routine **pccardDosDevCreate( )** in **pccardLib.c** uses this routine for that function. This should be invoked each time a new disk is inserted on media where the device geometry could possibly change. This function will re-read all device geometry data from the block driver, carve out and initialize all cache descriptors and blocks.

RETURNS **OK** or **ERROR** if the device is invalid or if the device geometry is invalid (**EINVAL**) or if there is not enough memory to perform the operation.

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

dcacheCbio

dcacheDevTune( )

NAME

dcacheDevTune( ) – modify tunable disk cache parameters

SYNOPSIS

```
STATUS dcacheDevTune
    (
    CBIO_DEV_ID dev,           /* device handle */
    int         dirtyMax,      /* max # of dirty cache blocks allowed */
    int         bypassCount,   /* request size for bypassing cache */
    int         readAhead,     /* how many blocks to read ahead */
    int         syncInterval   /* how many seconds between disk updates */
    )
```

DESCRIPTION

This function allows the user to tune some disk cache parameters to obtain better performance for a given application or workload pattern. These parameters are checked for sanity before being used, hence it is recommended to verify the actual parameters being set with **dcacheShow( )**.

Following is the description of each tunable parameter:

bypassCount
In order to achieve maximum performance, Disk Cache is bypassed for very large requests. This parameter sets the threshold number of blocks for bypassing the cache, resulting usually in the data being transferred by the low level driver directly to/from application data buffers (also known as cut-through DMA). Passing the value of 0 in this argument preserves the previous value of the associated parameter.

syncInterval
The Disk Cache provides a low priority task that will update all modified blocks onto the disk periodically. This parameters controls the time between these updates in

seconds. The longer this period, the better throughput is likely to be achieved, while risking to loose more data in the event of a failure. For removable devices this interval is fixed at 1 second. Setting this parameter to 0 results in immediate writes to disk when requested, resulting in minimal data loss risk at the cost of somewhat degraded performance.

readAhead
In order to avoid accessing the disk in small units, the Disk Cache will read many contiguous blocks once a block which is absent from the cache is needed. Increasing this value increases read performance, but a value which is too large may cause blocks which are frequently used to be removed from the cache, resulting in a low Hit Ratio, and increasing the number of Seeks, slowing down performance dramatically. Passing the value of 0 in this argument preserves the pervious value of the associated parameter.

dirtyMax
Routinely the Disk Cache will keep modified blocks in memory until it is specifically instructed to update these blocks to the disk, or until the specified time interval between disk updates has elapsed, or until the number of modified blocks is large enough to justify an update. Because the disk is updated in an ordered manner, and the blocks are written in groups when adjacent blocks have been modified, a larger dirtyMax parameter will minimize the number of Seek operation, but a value which is too large may decrease the Hit Ratio, thus degrading performance. Passing the value of 0 in this argument preserves the pervious value of the associated parameter.

RETURNS
OK or **ERROR** if device handle is invalid. Parameter value which is out of range will be silently corrected.

ERRNO
Not Available

SEE ALSO
dcacheCbio, **dcacheShow( )**

dcacheHashTest( )

NAME
dcacheHashTest( ) – test hash table integrity

SYNOPSIS
```
void dcacheHashTest
    (
    CBIO_DEV_ID dev
    )
```

DESCRIPTION
none

RETURNS
Not Available

ERRNO Not Available

SEE ALSO **dcacheCbio**

dcacheShow( )

NAME **dcacheShow( )** – print information about disk cache

SYNOPSIS

```
void dcacheShow
    (
    CBIO_DEV_ID dev,     /* device handle */
    int         verbose  /* 1 - display state of each cache block */
    )
```

DESCRIPTION This routine displays various information regarding a disk cache, namely current disk parameters, cache size, tunable parameters and performance statistics. The information is displayed on the standard output.

The *dev* argument is the device handle, if it is **NULL**, all disk caches are displayed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dcacheCbio**

devs( )

NAME **devs( )** – list all system-known devices

SYNOPSIS

```
void devs (void)
```

DESCRIPTION This command displays a list of all devices known to the I/O system.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **iosDevShow( )**, the VxWorks programmer guides.

dirList( )

NAME

dirList( ) – list contents of a directory (multi-purpose)

SYNOPSIS

```
STATUS dirList
    (
    int          fd,         /* file descriptor to write on */
    const char * dirString,  /* name of the directory to be listed */
    BOOL         doLong,     /* if TRUE, do long listing */
    BOOL         doTree      /* if TRUE, recurse into subdirs */
    )
```

DESCRIPTION

This command is similar to UNIX ls. It lists the contents of a directory in one of two formats. If *doLong* is **FALSE**, only the names of the files (or subdirectories) in the specified directory are displayed. If *doLong* is **TRUE**, then the file name, size, date, and time are displayed. If *doTree* flag is **TRUE**, then each subdirectory encountered will be listed as well (i.e. the listing will be recursive).

The *dirName* parameter specifies the directory to be listed. If *dirName* is omitted or **NULL**, the current working directory will be listed. *dirName* may contain wildcard characters to list some of the directory's contents.

LIMITATIONS

- With **dosFsLib** file systems, MS-DOS volume label entries are not reported.
- Although an output format very similar to UNIX "ls" is employed, some information items have no particular meaning on some file systems.
- Some file systems which do not support the POSIX compliant **dirLib( )** interface, can not support the *doLong* and *doTree* options.

RETURNS

OK or **ERROR**.

ERRNO

Not Available

SEE ALSO

usrFsLib, **dirLib**, **ls( )**, **ll( )**, **lsr( )**, **llr( )**, the VxWorks programmer guides.

diskFormat( )

NAME

diskFormat( ) – format a disk with dosFs

SYNOPSIS

```
STATUS diskFormat
    (
    const char * pDevName  /* name of the device to initialize */
    )
```

DESCRIPTION

This command in now obsolete. Use dosfsDiskFormat or **dosFsVolFormat( )** instead

This command formats a disk and creates the dosFs file system on it. The device must already have been created by the device driver and dosFs format component must be included.

EXAMPLE

```
    -> diskFormat "/fd0"
```

RETURNS

OK, or **ERROR** if the device cannot be opened or formatted.

ERRNO

Not Available

SEE ALSO

usrFsLib, **dosFsLib**, the VxWorks programmer guides.

diskInit( )

NAME

diskInit( ) – initialize a file system on a block device

SYNOPSIS

```
STATUS diskInit
    (
    const char *pDevName  /* name of the device to initialize */
    )
```

DESCRIPTION

This function is now obsolete.

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

usrFsLib

dosFsCacheCreate( )

NAME

dosFsCacheCreate( ) – create cache for a DosFS volume

SYNOPSIS

```
STATUS dosFsCacheCreate
    (
    char * volName,         /* volume name */
    char * dataCacheAddr,   /* memory address (NULL = KHEAP_ALLOC) */
    u_int  dataCacheSize,   /* size of cache */
    char * dirCacheAddr,    /* memory address (NULL = KHEAP_ALLOC) */
    u_int  dirCacheSize,   /* size of cache */
    char * fatCacheAddr,   /* memory address (NULL = KHEAP_ALLOC) */
    u_int  fatCacheSize    /* size of cache */
    )
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsCacheDelete( )

NAME **dosFsCacheDelete( )** – delete the disk cache for a dosFs volume

SYNOPSIS

```
STATUS dosFsCacheDelete
    (
    const char * volName  /* dosFs volume name */
    )
```

DESCRIPTION This routine removes the disk cache for *volName* and frees the allocated memory if it was requested from the system memory pool.

RETURNS STATUS.

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsCacheInfo( )

NAME **dosFsCacheInfo( )** – retrieve a cache's settings

SYNOPSIS

```
STATUS dosFsCacheInfo
    (
    char *            volName,   /* Name of the DosFS volume */
    DOS_CACHE_TYPE    type,      /* Identify which cache to tune */
    DOS_CACHE_INFO *  pSettings  /* Cache settings */
    )
```

DESCRIPTION

This routine allows the user to retrieve the cache's **bypass** and **readAhead** parameters. Reading more than **bypass** sectors at once may trigger a cache flush and data will be retrieved directly from disk instead of cache. When reading data, DosFS will try to read **readAhead** sectors at a time.

volName is a **NULL** terminated character string identifying the name of the DosFS device. *type* identifies the volume's cache for which to retrieve information. Valid values are **DOS_DATA_CACHE**, **DOS_DIR_CACHE** and **DOS_FAT_CACHE**. *pSettings* points to a structure containing parameters for the **bypass** and **readAhead** values of the cache.

RETURNS

OK on success, **ERROR** otherwise

ERRNO

Not Available

SEE ALSO

dosFsCacheLib

dosFsCacheLibInit( )

NAME

dosFsCacheLibInit( ) – initialize dosFsCache library.

SYNOPSIS

```
void dosFsCacheLibInit
    (
    u_int defaultDataCacheSize,
    u_int defaultDirCacheSize,
    u_int defaultFatCacheSize
    )
```

DESCRIPTION

none

RETURNS

N/A.

/NOMANUAL

ERRNO

Not Available

SEE ALSO

dosFsCacheLib

dosFsCacheOptionsGet( )

NAME

dosFsCacheOptionsGet( ) – get this dosFs volume's disk cache options

SYNOPSIS

```
UINT dosFsCacheOptionsGet
    (
    char * volName  /* dosFs volume name */
    )
```

DESCRIPTION

This routine gets the cache options for the dosFs volume *volName*.

RETURNS

value of the volume's options.

ERRNO

Not Available

SEE ALSO

dosFsCacheLib

dosFsCacheOptionsSet( )

NAME

dosFsCacheOptionsSet( ) – set this dosFs volume's disk cache options

SYNOPSIS

```
UINT dosFsCacheOptionsSet
    (
    char * volName,  /* dosFs volume name */
    UINT   options   /* new options */
    )
```

DESCRIPTION

This routine sets the cache options for the dosFs volume *volName*. Currently the only option available is **DOS_CACHE_VOL_NO_DMA**, which means that **dosFsCacheLib** does not need to use DMA-safe buffers on *volName* when large transfers to and from the media are requested. This option can gain performance on large transfers, and is recommended for USB devices.

RETURNS

new value of the volume's options.

ERRNO

Not Available

SEE ALSO

dosFsCacheLib

dosFsCacheShow( )

NAME

dosFsCacheShow( ) – show information regarding a dosFs volume's cache

SYNOPSIS

```
void dosFsCacheShow
    (
    const char * volName,  /* volume name */
    u_int        level     /* verbosity level */
    )
```

DESCRIPTION

This routine displays information regarding an specific dosFs volume *volName* cache. If *level* is zero, a summary is displayed, otherwise the current state of the internal hash table and LRU list is displayed.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

dosFsShow

dosFsCacheTune( )

NAME

dosFsCacheTune( ) – tune a cache's settings

SYNOPSIS

```
STATUS dosFsCacheTune
    (
    char *           volName,   /* Name of the DosFS volume */
    DOS_CACHE_TYPE   type,      /* Identify which cache to tune */
    DOS_CACHE_INFO * pSettings  /* New cache settings */
    )
```

DESCRIPTION

This routine allows the user to tune the cache's **bypass** and **readAhead** parameters. Reading more than **bypass** sectors at once may trigger a cache flush and data will be retrieved directly from disk instead of cache. When reading data, DosFS will try to read **readAhead** sectors at a time.

volName is a **NULL** terminated character string identifying the name of the DosFS device. *type* identifies which cache associated with the volume to tune. Valid values are **DOS_DATA_CACHE**, **DOS_DIR_CACHE** and **DOS_FAT_CACHE**. *pSettings* is a structure containing parameters for the **bypass** and **readAhead** values of the cache.

RETURNS

OK on success, **ERROR** otherwise

ERRNO
Not Available

SEE ALSO
dosFsCacheLib

dosFsChkDsk( )

NAME
dosFsChkDsk( ) – make volume integrity checking.

SYNOPSIS
```
STATUS dosFsChkDsk
    (
    FAST DOS_FILE_DESC_ID pFd,    /* file descriptor of root dir */
    u_int                 params  /* check level and verbosity */
    )
```

DESCRIPTION
This library does not make integrity check process itself, but instead uses routine provided by **dosChkLib**. This routine prepares parameters and invokes checking routine via preinitialized function pointer. If **dosChkLib** is not configured into vxWorks, this routine returns **ERROR**.

Ownership on device should be taken by an upper level routine.

RETURNS
STATUS as returned by volume checking routine or
ERROR, if such routine is not installed.

ERRNO
S_dosFsLib_UNSUPPORTED.

SEE ALSO
dosFsLib

dosFsClose( )

NAME
dosFsClose( ) – close a dosFs file

SYNOPSIS
```
STATUS dosFsClose
    (
    DOS_FILE_DESC_ID pFd  /* file descriptor pointer */
    )
```

DESCRIPTION
This routine closes the specified dosFs file. If file contains excess clusters beyond **EOF** they are freed, when last file descriptor is being closed for that file.

RETURNS **OK**, or **ERROR** if directory couldn't be flushed or entry couldn't be found.

ERRNO **S_dosFsLib_INVALID_PARAMETER**
S_dosFsLib_DELETED
S_dosFsLib_FD_OBSOLETE /NOMANUAL

SEE ALSO **dosFsLib**

dosFsDefaultCacheSizeSet( )

NAME **dosFsDefaultCacheSizeSet( )** – set the default disk cache size

SYNOPSIS

```
void dosFsDefaultCacheSizeSet
    (
    UINT newDataDefaultSize,  /* new default size for dosFs data cache */
    UINT newDirDefaultSize,   /* new default size for dosFs dir cache */
    UINT newFatDefaultSize    /* new default size for dosFs FAT cache */
    )
```

DESCRIPTION This routine sets the default disk cache size to be used for the next dosFs instantiations.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsDefaultDataCacheSizeGet( )

NAME **dosFsDefaultDataCacheSizeGet( )** – get the default data cache size

SYNOPSIS

```
UINT dosFsDefaultDataCacheSizeGet (void)
```

DESCRIPTION This routine gets the default data cache size.

RETURNS value of the default data cache size.

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsDefaultDirCacheSizeGet( )

NAME **dosFsDefaultDirCacheSizeGet( )** – get the default directory cache size

SYNOPSIS

```
UINT dosFsDefaultDirCacheSizeGet (void)
```

DESCRIPTION This routine gets the default directory cache size.

RETURNS value of the default directory cache size.

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsDefaultFatCacheSizeGet( )

NAME **dosFsDefaultFatCacheSizeGet( )** – get the default FAT cache size

SYNOPSIS

```
UINT dosFsDefaultFatCacheSizeGet (void)
```

DESCRIPTION This routine gets the default FAT cache size.

RETURNS value of the default FAT cache size.

ERRNO Not Available

SEE ALSO **dosFsCacheLib**

dosFsDevCreate( )

NAME **dosFsDevCreate( )** – create file system device.

SYNOPSIS

```
STATUS dosFsDevCreate
    (
    char *   pDevName,            /* device name */
    device_t device,              /* underlying XBD block device */
    u_int    maxFiles,            /* max no. of simultaneously open files */
    int      dosDevCreateOptions  /* write option & volume integrity */
    )
```

DESCRIPTION

This routine associates an XBD device with a logical I/O device name and prepare it to perform file system functions. It takes an XBD device handle, typically created by **xbdBlkDevCreate( )** or **xbdPartitionDevCreate( )**, and defines it as a dosFs volume. As a result, when high-level I/O operations (e.g., **open( )**, **write( )**) are performed on the device, the calls will be routed through **dosFsLib**. The *device* parameter is the handle of the underlying partition or block device XBD.

The argument *maxFiles* specifies the number of files that can be opened at once on the device.

The volume structure integrity can be automatically checked during volume mounting. Parameter *dosDevCreateOptions* defines checking level (**DOS_CHK_ONLY** or **DOS_CHK_REPAIR**), that can be bitwise or-ed with check verbosity level value (**DOS_CHK_VERB_SILENT**, **DOS_CHK_VERB_1** or **DOS_CHK_VERB_2**).

If the value of *dosDevCreateOptions* is 0, the default checking level is used. The default level is (**DOS_CHK_ONLY** | **DOS_CHK_VERB_2**).

To suppress the automatic check disk, bitwise or (**DOS_CHK_NONE**) or set *dosDevCreateOptions* to NONE.

Disk checking is normally suppressed on volumes marked clean. To force a disk-check, bitwise or (**DOS_CHK_FORCE**).

The volume may be configured to request **DOS_WRITE_THROUGH** writes for some or all of the disk operations. Additional bits of parameter *dosDevCreateOptions* define the volume's write-through setting. The default (zero) is to use copyback writes (**DOS_WRITE**) for all write operations. The default is the fastest configuration.

To writethrough all FAT table write operations, or in **DOS_WRITE_THROUGH_FAT** To writethrough all directory entry write operations, or in **DOS_WRITE_THROUGH_DIR** To writethrough all user data buffers, or in **DOS_WRITE_THROUGH_USR** To writethrough both FAT and DIRENT operations, or them together. (**DOS_WRITE_THROUGH_DIR** | **DOS_WRITE_THROUGH_FAT** | **DOS_CHK_NONE**)

User data writes will still use copyback **DOS_WRITE** operations when using (**DOS_WRITE_THROUGH_DIR** | **DOS_WRITE_THROUGH_FAT**)

To write-through all write operations, including all user data, or in **DOS_WRITE_THROUGH_ALL**. This is the slowest operation and all write operation made by the file system will be **DOS_WRITE_THROUGH**.

To enable Unicode filenames, or in **DOS_FILENAMES_UNICODE**. Case insensitivity (if enabled) currently applies only to ASCII values even when Unicode is turned on. For instance, a German eszet is never considered the same as two uppercase S characters, but two uppercase S characters can match two lowercase S characters because these are both in the first 128 character codes.

NOTE

Setting parameter *dosDevCreateOptions* to NONE (-1) will both disable the automated chkdsk and force copyback (**DOS_WRITE**) operation. Unicode filenames will not be enabled.

Note that during a call to **dosFsDevCreate( )** actual disk accesses are deferred to the time when **open( )** or **creat( )** are first called. That is also when the automatic disk checking will take place. Therefore this function will succeed in cases where a removable disk is not present in the drive.

RETURNS **OK**, or **ERROR** if the device name is already in use or insufficient memory.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsDevDelete( )

NAME **dosFsDevDelete( )** – delete a dosFs volume

SYNOPSIS

```
STATUS dosFsDevDelete
    (
    DOS_VOLUME_DESC_ID pVolDesc  /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine deletes a dosFs volume.

RETURNS **OK** on success, **ERROR** otherwise

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsDiskProbe( )

NAME **dosFsDiskProbe( )** – probe if a device contains a valid dosFs

SYNOPSIS

```
STATUS dosFsDiskProbe
    (
    device_t xbdDevice  /* XBD device to probe */
    )
```

DESCRIPTION This routine probes if a device (or a partition) contains a valid DOS FS.

RETURNS **OK** if successful. **ERROR** otherwise.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsFdFree( )

NAME **dosFsFdFree( )** – free a file descriptor

SYNOPSIS

```
void dosFsFdFree
    (
    DOS_FILE_DESC_ID pFd
    )
```

DESCRIPTION This routine marks a file descriptor as free and decreases reference count of a referenced file handle.

RETURNS N/A.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsFdGet( )

NAME **dosFsFdGet( )** – get an available file descriptor

SYNOPSIS

```
DOS_FILE_DESC_ID dosFsFdGet
    (
    DOS_VOLUME_DESC_ID pVolDesc
    )
```

DESCRIPTION This routine obtains a free dosFs file descriptor.

RETURNS Pointer to file descriptor, or **NULL**, if none available.

ERRNO **S_dosFsLib_NO_FREE_FILE_DESCRIPTORS** /NOMANUAL

SEE ALSO **dosFsLib**

dosFsFmtLibInit( )

NAME **dosFsFmtLibInit( )** – initialize the MS-DOS formatting library

SYNOPSIS

```
void dosFsFmtLibInit(void)
```

DESCRIPTION This function is called to optionally enable the formatting functionality from **dosFsLib**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dosFsFmtLib**, **dosFsLib**, /NOMANUAL

dosFsFmtTest( )

NAME **dosFsFmtTest( )** – UNITEST CODE

SYNOPSIS

```
void dosFsFmtTest
    (
    int size
    )
```

DESCRIPTION /NOMANUAL

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dosFsFmtLib**

dosFsHdlrInstall( )

NAME

dosFsHdlrInstall( ) – install handler.

SYNOPSIS

```
STATUS dosFsHdlrInstall
    (
    DOS_HDLR_DESC_ID hdlrsList,  /* appropriate list */
    DOS_HDLR_DESC_ID hdlr        /* ptr on handler descriptor */
    )
```

DESCRIPTION

This library does not directly access directory structure, nor FAT, rather it uses particular handlers to serve such accesses. This function is intended for use by the **dosFsLib** sub-modules only.

This routine installs a handler into DOS FS handlers list. There are two such lists: FAT Handlers List (*dosFatHdlrsList*) and Directory Handlers List (*dosDirHdlrsList*). Each handler must provide its unique Id (see **dosFsLibP.h**) and pointer to appropriate list to install it to. All lists are sorted by Id-s in ascending order. Every handler is tried to be mounted on each new volume in accordance to their order in list, until succeeded. So preferable handlers, that supports the same type of volumes must have less Id values.

RETURNS

STATUS.

/NOMANUAL

ERRNO

Not Available

SEE ALSO

dosFsLib

dosFsIoctl( )

NAME

dosFsIoctl( ) – do device specific control function

SYNOPSIS

```
STATUS dosFsIoctl
    (
    FAST DOS_FILE_DESC_ID pFd,       /* fd of file to control */
    int                   function,  /* function code */
    int                   arg        /* some argument */
    )
```

DESCRIPTION

This routine performs the following ioctl functions.

Any ioctl function codes, that are not supported by this routine are passed to the underlying XBD module for handling.

There are some **ioctl( )** functions, that suppose to receive as result a 32-bit numeric value (FIONFREE, FIOWHERE and so on), however disks and files with size grater, than 4GB are supported. In order to solve this contradiction new **ioctl( )** functions are provided. They have the same name as basic functions, but with suffix **64**: FIONFREE64, FIOWHERE64 and so on. These functions gets pointer to **long long** as an argument. Also FIOWHERE64 returns value via argument, but not as **ioctl( )**returned value. If an ioctl fails, the task's status (see **errnoGet( )**) indicates the nature of the error.

RETURNS

OK or current position in file for FIOWHERE, or **ERROR** if function failed or driver returned error, or if function supposes 32 bit result value, but actual result overloads this restriction.

ERRNO

S_dosFsLib_INVALID_PARAMETER
S_dosFsLib_VOLUME_NOT_AVAILABLE
S_dosFsLib_FD_OBSOLETE
S_dosFsLib_DELETED
S_dosFsLib_32BIT_OVERFLOW /NOMANUAL

SEE ALSO

dosFsLib

dosFsLastAccessDateEnable( )

NAME

dosFsLastAccessDateEnable( ) – enable last access date updating for this volume

SYNOPSIS

```
STATUS dosFsLastAccessDateEnable
    (
    DOS_VOLUME_DESC_ID dosVolDescId,  /* dosfs volume ID to alter */
    BOOL               enable         /* TRUE = enable update, FALSE =
disable update */
    )
```

DESCRIPTION

This function enables or disables updating of the last access date directory entry field on open-read-close operations for the given dosFs volume. The last access date file indicates the last date that a file has been read or written. When the optional last access date field update is enabled, read operations on a file will cause a write to the media.

RETURNS

OK or **ERROR** if the volume is invalid or enable is not **TRUE** or **FALSE**.

ERRNO

Not Available

SEE ALSO

dosFsLib

dosFsLibInit( )

NAME **dosFsLibInit( )** – prepare to use the dosFs library

SYNOPSIS
```
STATUS dosFsLibInit
    (
    int maxFiles,
    int options
    )
```

DESCRIPTION This routine initializes the dosFs library. This initialization is enabled when the configuration macro **INCLUDE_DOSFS** is defined. This routine installs **dosFsLib** as a driver in the I/O system driver table, and allocates and sets up the necessary structures. The driver number assigned to **dosFsLib** is placed in the global variable *dosFsDrvNum*.

RETURNS **OK** or **ERROR**, if driver can not be installed.

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsMonitorDevCreate( )

NAME **dosFsMonitorDevCreate( )** – create a dosFs volume through the fs monitor

SYNOPSIS
```
STATUS dosFsMonitorDevCreate
    (
    device_t xbdId,    /* XBD for the device on which to mount. */
    char *   pDevName  /* Name of the DOS FS device (mount point). */
    )
```

DESCRIPTION This routine creates an DOS FS device.

RETURNS **OK** if successful. **ERROR** otherwise.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsOpen( )

NAME **dosFsOpen( )** – open a file on a dosFs volume

SYNOPSIS

```
DOS_FILE_DESC_ID dosFsOpen
    (
    DOS_VOLUME_DESC_ID pVolDesc,  /* pointer to volume descriptor */
    char *             pPath,     /* dosFs full path/filename */
    int                flags,     /* file open flags */
    int                mode       /* file open permissions (mode) */
    )
```

DESCRIPTION This routine opens the file *name* with the specified mode (**O_RDONLY**/**O_WRONLY**/**O_RDWR**/CREATE/TRUNC). The directory structure is searched, and if the file is found a dosFs file descriptor is initialized for it. Extended flags are provided by DOS FS for more efficiency:

- **DOS_O_CONTIG_CHK** - to check file for contiguity.
- **DOS_O_CASENS** - force the file name lookup in case insensitive manner, (if directory format provides such opportunity)

If this is the very first open for the volume, configuration data will be read from the disk automatically (via **dosFsVolMount( )**).

RETURNS A pointer to a dosFs file descriptor, or **ERROR** if the volume is not available, or there are no available dosFs file descriptors, or there is no such file and **O_CREAT** was not specified, or file can not be opened with such permissions.

ERRNO **S_dosFsLib_INVALID_PARAMETER**
S_dosFsLib_READ_ONLY
S_dosFsLib_FILE_NOT_FOUND
S_dosFsLib_FILE_EXISTS /NOMANUAL

SEE ALSO **dosFsLib**

dosFsShow( )

NAME **dosFsShow( )** – display dosFs volume configuration data.

SYNOPSIS

```
STATUS dosFsShow
    (
    void * pDevName,  /* name of device */
    u_int   level       /* detail level */
    )
```

DESCRIPTION

This routine obtains the dosFs volume configuration for the named device, formats the data, and displays it on the standard output.

If no device name is specified, the current default device is described.

RETURNS

OK or **ERROR**, if no valid device specified.

ERRNO

Not Available

SEE ALSO

dosFsShow

dosFsVolDescGet( )

NAME

dosFsVolDescGet( ) – convert a device name into a DOS volume descriptor pointer.

SYNOPSIS

```
DOS_VOLUME_DESC_ID dosFsVolDescGet
    (
    void *    pDevNameOrPVolDesc,  /* device name or pointer to dos vol desc
*/
    u_char ** ppTail               /* return ptr for name, used in iosDevFind
*/
    )
```

DESCRIPTION

This routine validates *pDevNameOrPVolDesc* to be a DOS volume descriptor pointer else a path to a DOS device. This routine uses the standard **iosLib** function **iosDevFind( )** to obtain a pointer to the device descriptor. If device is eligible, *ppTail* is filled with the pointer to the first character following the device name. Note that ppTail is passed to **iosDevFind( )**. *ppTail* may be passed as **NULL**, in which case it is ignored.

RETURNS

A **DOS_VOLUME_DESC_ID** or **NULL** if not a DOSFS device.

ERRNO

S_dosFsLib_INVALID_PARAMETER

SEE ALSO

dosFsLib

dosFsVolFormat( )

NAME **dosFsVolFormat( )** – format an MS-DOS compatible volume

SYNOPSIS

```
STATUS dosFsVolFormat
    (
    char *  path,        /* path for volume to format */
    int     opt,         /* bit-wise or'ed options */
    FUNCPTR pPromptFunc  /* interactive parameter change callback */
    )
```

DESCRIPTION This utility routine performs the initialization of file system data structures on a disk. It supports FAT12 for small disks, FAT16 for medium size and FAT32 for large volumes. The *device* argument is a device name known to the I/O system.

The *opt* argument is a bit-wise or'ed combination of options controlling the operation of this routine as follows:

DOS_OPT_DEFAULT
If the current volume boot block is reasonably intact, use existing parameters, else calculate parameters based only on disk size, possibly reusing only the volume label and serial number.

DOS_OPT_PRESERVE
Attempt to preserve the current volume parameters even if they seem to be somewhat unreliable.

DOS_OPT_BLANK
Disregard the current volume parameters, and calculate new parameters based only on disk size.

DOS_OPT_QUIET
Do not produce any diagnostic output during formatting.

DOS_OPT_FAT16
Format the volume with FAT16. Valid on volumes up to 2 Gbytes big. For larger ones, FAT32 must be used.

DOS_OPT_FAT32
Format the volume with FAT32, even if the disk is smaller than 2 Gbytes, but is larger then 512 Mbytes.

DOS_OPT_VXLONGNAMES
Note that this option is deprecated. Calling dosFsVolFormat with this option will result in an error.

The third argument, *pPromptFunc* is an optional pointer to a function that may interactively prompt the user to change any of the modifiable volume parameters before formatting:

```
void formatPromptFunc( DOS_VOL_CONFIG *pConfig );
```

The **pConfig* structure upon entry to **formatPromptFunc( )** will contain the initial volume parameters, some of which can be changed before it returns. *pPromptFunc* should be **NULL** if no interactive prompting is required.

COMPATIBILITY Although this routine tries to format the disk to be compatible with Microsoft implementations of the FAT and FAT32 file systems, there may be differences which are not under WRS control. For this reason, it is highly recommended that any disks which are expected to be interchanged between vxWorks and Windows should be formatted under Windows to provide the best interchangeability. The WRS implementation is more flexible, and should be able to handle the differences when formatting is done on Windows, but Windows implementations may not be able to handle minor differences between their implementation and ours.

AVAILABILITY This function 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.

RETURNS **OK** or **ERROR** if was unable to format the disk.

ERRNO Not Available

SEE ALSO **dosFsFmtLib**

dosFsVolFormatFd( )

NAME **dosFsVolFormatFd( )** – format an MS-DOS compatible volume via an opened FD

SYNOPSIS

```
STATUS dosFsVolFormatFd
    (
    int      fd,           /* rawFs file descriptor */
    int      opt,          /* bit-wise or'ed options */
    FUNCPTR pPromptFunc  /* interactive parameter change callback */
    )
```

DESCRIPTION Refer to **dosFsVolFormat( )**'s documentation. It should also be noted that the file descriptor parameter will be closed regardless of the routine's outcome.

RETURNS **OK** or **ERROR** if was unable to format the disk.

ERRNO Not Available

SEE ALSO **dosFsFmtLib**

dosFsVolIsFat12( )

NAME

dosFsVolIsFat12( ) – determine if a MSDOS volume is FAT12 or FAT16

SYNOPSIS

```
int dosFsVolIsFat12
    (
    u_char * pBootBuf  /* boot parameter block buffer */
    )
```

DESCRIPTION

This routine is the container for the logic which determines if a dosFs volume is using FAT12 or FAT16. Two methods are implemented. Both methods use information from the volumes boot parameter block fields found in the boot sector.

The first FAT determination method follows the recommendations outlined in the Microsoft document:

"Hardware White Paper
 Designing Hardware for Microsoft Operating Systems
 FAT: General Overview of On-Disk Format
 Version 1.02, May 5, 1999
 Microsoft Corporation"

This method is used in the hopes that greater compatability with MSDOS formatted media will be achieved. The Microsoft recommended method for FAT type determination between FAT12 and FAT16 is done via the count of clusters on the volume.

The Microsoft recommended approach is as follows:

1.) Determine the count of sectors occupied by the root directory
   entries for this volume, rounding up:

rootDirSecs = ((rootEntCount * dirEntSz) + (bytesPerSec-1)) / bytesPerSec;

Where dirEntSz is 32 for MSDOS 8.3, and 64 for VXLONGNAMES.

2.) Determine the count of sectors occupied by the volumes data region:

dataRgnSecs = totalSecs - (reservedSecs + (nFats * fatSecs) + rootDirSecs);

3.) determine the count of clusters, rounding down:

countOfClusts = dataSecs / secsPerClust; /* Note: this rounds down. */

Note: countOfClusts represents the count of data clusters, starting at two.

4.) determine the FAT types based on the count of clusters on the volume,

```
  if (countOfClusts < 4085) /* Microsoft recommends using "less than" */
      {
      /* Volume is FAT12 */
```

```
    }
  else
    {
    /* Volume is FAT16 */
    }
```

An alternate method is used when mounting a known VxWorks DOSFS-1.0 volume. This method is used for greater backward compatability with VxWorks DOSFS-1.0 volumes. See also: SPR#34704. The VxWorks dosFs1 method deviates from the Microsoft currently recommened method.

This is the VxWorks DOSFS1 method per **dosFsVolDescFill( )**, **dosFsLib.c**, dosFs 1.0, revision history: "03l,16mar99,dgp". Using the identical method here will help ensure backward compatablitity when mounting volumes formatted by the VxWorks dosFs1.0 code.

The VxWorks DOSFS 1.0 approach is as follows:

1.) Get starting sector of the root directory:

rootSec = reservedSecs + (nFats * secsPerFat);

2.) Get the size of the root dir in bytes:

rootBytes = (nRootEnts * dirEntSz):

Where dirEntSz is 32 for MSDOS 8.3, and 64 for VXLONGNAMES.

3.) Get the starting sector of the data area:

dataSec = rootSec + ((rootBytes + bytesPerSec-1) / bytesPerSec);

4.) Get the number of "FAT entries":

```
countOfClusts =
    ( ( (totalSecs - dataSecs) / secsPerClust) + DOS_MIN_CLUST);
```

5.) Choose the FAT type based on the count of clusters, note DOSFS1 uses less than or equal here.

```
if (countOfClusts <= 4085) /* VxDosFs1 uses less than or equal to. */
  {
  /* use FAT12 */
  }
else
  {
  /* use FAT16 */
  }
```

By mimicking the dosFs 1.0 approach, we should be able to mount all dosFs 1.0 volumes correctly. By using the microsoft recommened approach in all other cases, we should be as compatable as possible with Microsoft OS's.

The volumes Boot Parameter Block fields MUST be validated for sanity before this routine is called.

pBootBuf is not verified, DO NOT pass this routine a **NULL** pointer. This routine is also used by **dosFsFmtLib.c**

RETURNS **TRUE** if the FAT type is FAT12, **FALSE** if the FAT type is FAT16, or **ERROR** if the data is invalid.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsVolUnmount( )

NAME **dosFsVolUnmount( )** – unmount a dosFs volume

SYNOPSIS

```
STATUS dosFsVolUnmount
    (
    void * pDevNameOrPVolDesc  /* device name or ptr to */
                               /* volume descriptor */
    )
```

DESCRIPTION This routine is called when I/O operations on a volume are to be discontinued. This is the preferred action prior to changing a removable disk.

All buffered data for the volume is written to the device (if possible, with no error returned if data cannot be written), any open file descriptors are marked as obsolete, and the volume is marked as not currently mounted.

When a subsequent **open( )** operation is initiated on the device, new volume will be mounted automatically.

Once file descriptors have been marked as obsolete, any attempt to use them for file operations will return an error. (An obsolete file descriptor may be freed by using **close( )**. The call to **close( )** will return an error, but the descriptor will in fact be freed).

This routine may also be invoked by calling **ioctl( )** with the FIOUNMOUNT function code.

This routine must not be called from interrupt level.

RETURNS **OK**, or **ERROR** if the volume was not mounted.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsVolumeOptionsGet( )

NAME **dosFsVolumeOptionsGet( )** – get this volume's disk options

SYNOPSIS

```
UINT dosFsVolumeOptionsGet
    (
    char * volName  /* dosFs volume name */
    )
```

DESCRIPTION This routine gets the volume options for the dosFs volume *volName*. It replaces the routine **dosFsCacheOptionsGet( )**.

volName is a **NULL** terminated character string identifying the DosFS volume.

RETURNS **DOS_VOLUME_VOL_NO_DMA** if that option is enabled;
0 if no options are enabled; **ERROR** on failure

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsVolumeOptionsSet( )

NAME **dosFsVolumeOptionsSet( )** – set this volume's disk options

SYNOPSIS

```
UINT dosFsVolumeOptionsSet
    (
    char * volName,  /* dosFs volume name */
    UINT   options   /* new options */
    )
```

DESCRIPTION

This routine sets the volume options for the dosFs volume *volName*. Currently the only option available is **DOS_VOLUME_VOL_NO_DMA**, which means that DosFS does not need to use DMA-safe buffers on *volName* when large transfers to and from the media are requested. This option may gain performance on large transfers, and is recommended for USB devices. It replaces the routine **dosFsCacheOptionsSet( )**.

volName is a **NULL** terminated character string that corresponds to the name of the desired DosFS volume. Valid settings for *options* are currently **DOS_VOLUME_VOL_NO_DMA** or 0.

RETURNS

new value of the volume's options, or **ERROR** on failure.

ERRNO

Not Available

SEE ALSO

dosFsLib

dosFsXbdBlkCopy( )

NAME

dosFsXbdBlkCopy( ) – copy blocks on the underlying XBD block device.

SYNOPSIS

```
STATUS dosFsXbdBlkCopy
    (
    DOS_VOLUME_DESC * pVolDesc,  /* volume descriptor */
    sector_t          srcBlock,
    sector_t          dstBlock,
    sector_t          numBlocks
    )
```

DESCRIPTION

none

RETURNS

STATUS.

/NOMANUAL

ERRNO

Not Available

SEE ALSO

dosFsLib

dosFsXbdBlkRead( )

NAME **dosFsXbdBlkRead( )** – read blocks from the underlying XBD block device.

SYNOPSIS

```
STATUS dosFsXbdBlkRead
    (
    DOS_VOLUME_DESC * pVolDesc,    /* volume descriptor */
    sector_t          startBlock,  /* starting block of transfer */
    sector_t          numBlocks,   /* number of blocks to transfer */
    addr_t            buffer       /* address of the memory buffer */
    )
```

DESCRIPTION none

RETURNS STATUS.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsXbdBlkWrite( )

NAME **dosFsXbdBlkWrite( )** – write blocks to the underlying XBD block device.

SYNOPSIS

```
STATUS dosFsXbdBlkWrite
    (
    DOS_VOLUME_DESC * pVolDesc,    /* volume descriptor */
    sector_t          startBlock,  /* starting block of write */
    sector_t          numBlocks,   /* number of blocks to write */
    addr_t            buffer,      /* address of the memory buffer */
    DOS_RW            operation    /* DOS_WRITE/DOS_WRITE_THROUGH */
    )
```

DESCRIPTION none

RETURNS STATUS.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsXbdBytesRW( )

NAME **dosFsXbdBytesRW( )** – read/write bytes to/from the underlying XBD block device.

SYNOPSIS

```
STATUS dosFsXbdBytesRW
    (
    DOS_VOLUME_DESC * pVolDesc,    /* volume descriptor */
    sector_t          startBlock,  /* starting block of the transfer */
    off_t             offset,      /* offset into block in bytes */
    addr_t            buffer,      /* address of data buffer */
    size_t            nBytes,      /* number of bytes to transfer */
    u_int             operation    /* DOS_READ/DOS_WRITE/WRITE_THROUGH */
    )
```

DESCRIPTION none

RETURNS STATUS.

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosFsXbdIoctl( )

NAME **dosFsXbdIoctl( )** – Misc control operations

SYNOPSIS

```
STATUS dosFsXbdIoctl
    (
    DOS_VOLUME_DESC * pVolDesc,
    UINT32            command,
    addr_t            arg
    )
```

DESCRIPTION This performs the requested old CBIO **ioctl( )** operations.

RETURNS **OK** or **ERROR** and may otherwise set errno.

/NOMANUAL

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dosFsLib**

dosPathParse( )

NAME **dosPathParse( )** – parse a full pathname into an array of names.

SYNOPSIS

```
int dosPathParse
    (
    u_char *     path,
    PATH_ARRAY * pnamePtrArray,
    size_t       sizeArray
    )
```

DESCRIPTION This routine is similar to **pathParse( )**, but on the contrary it does not allocate additional buffers nor changes the path string.

Parses a path in directory tree which has directory names separated by **/** or ''s. It fills the supplied array of structures with pointers to directory and file names and correspondence name length. All occurrences of **//**, **.** and **..** are right removed from path. All tail dots and spaces are broken from each name, that is name like "abc. . ." is treated as just "abc".

For instance, "/usr/vw/data/../dir/file" gets parsed into

```
                         namePtrArray
                        |---------|
  ---------------------------o    |
  |                     |    3    |
  |                     |---------|
  |   -----------------------o    |
  |   |                 |    2    |
  |   |                 |---------|
  |   |          ------------o    |
  |   |          |      |    3    |
  |   |          |      |---------|
  |   |          |   --------o    |
  |   |          |   |  |    4    |
  |   |          |   |  |---------|
  v   v          v   v  |   NULL  |
  |   |          |   |  |    0    |
  |   |          |   |  |---------|
  v   v          v   v
 |------------------------|
 |usr/vw/data/../dir/file |
 |-------\-----/----------|
         ignored
```

Note that UTF-8 bytes that are not representing ASCII characters **.**, **/**, etc., never compare equal to **.**, **/**, etc., so that no special work is required here for Unicode.

In the future, the "../" erasure trick is likely to vanish. Do not rely on the fact that you can currently access "**nosuchdir/../file.txt**" when "nosuchdir" does not exist.

RETURNS number of levels in path.

ERRNO **S_dosFsLib_ILLEGAL_PATH**
S_dosFsLib_ILLEGAL_NAME

SEE ALSO **dosFsLib**

dosSetVolCaseSens( )

NAME **dosSetVolCaseSens( )** – set case sensitivity of volume

SYNOPSIS

```
STATUS dosSetVolCaseSens
    (
    DOS_VOLUME_DESC_ID pVolDesc,
    BOOL               sensitivity
    )
```

DESCRIPTION Pass **TRUE** to setup a case sensitive volume. Pass **FALSE** to setup a case insensitive volume. Note this affects rename lookups only.

RETURNS **TRUE** if pVolDesc pointed to a DOS volume.

ERRNO Not Available

SEE ALSO **dosFsLib**

dosfsDiskFormat( )

NAME **dosfsDiskFormat( )** – format a disk with dosFs

SYNOPSIS

```
STATUS dosfsDiskFormat
    (
    const char * pDevName  /* name of the device to initialize */
    )
```

DESCRIPTION This command formats a disk and creates the dosFs file system on it. The device must already have been created by the device driver and dosFs format component must be included.

EXAMPLE

```
    -> dosfsDiskFormat "/fd0"
```

RETURNS **OK**, or **ERROR** if the device cannot be opened or formatted.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dosFsLib**, the VxWorks programmer guides.

dosfsDiskToHost16( )

NAME **dosfsDiskToHost16( )** – convert uint16_t from on-disk to host format

SYNOPSIS

```
uint16_t  dosfsDiskToHost16
    (
    uint8_t * pSrc
    )
```

DESCRIPTION This routine converts a uint16_t from on-disk format to host's endian-ness.

RETURNS uint16_t in hosts's endian-ness

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosfsDiskToHost32( )

NAME **dosfsDiskToHost32( )** – convert uint32_t from on-disk to host format

SYNOPSIS

```
uint32_t  dosfsDiskToHost32
    (
    uint8_t * pSrc
    )
```

DESCRIPTION This routine converts a uint32_t from on-disk format to host's endian-ness.

RETURNS uint32_t in host's endian-ness

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosfsHostToDisk16( )

NAME **dosfsHostToDisk16( )** – convert uint16_t from host to on-disk format

SYNOPSIS

```
void  dosfsHostToDisk16
    (
    uint16_t  src,
    uint8_t * pDest
    )
```

DESCRIPTION This routine converts a uint16_t from host's memory to on-disk format.

RETURNS N/A

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dosfsHostToDisk32( )

NAME **dosfsHostToDisk32( )** – convert uint32_t from host to on-disk format

SYNOPSIS

```
void dosfsHostToDisk32
    (
    uint32_t  src,
    uint8_t * pDest
    )
```

DESCRIPTION This routine converts a uint32_t from host's memory to on-disk format.

RETURNS N/A

/NOMANUAL

ERRNO Not Available

SEE ALSO **dosFsLib**

dpartDevCreate( )

NAME **dpartDevCreate( )** – Initialize a partitioned disk

SYNOPSIS

```
CBIO_DEV_ID dpartDevCreate
    (
    CBIO_DEV_ID subDev,           /* lower level CBIO device */
    int         nPart,            /* # of partitions */
    FUNCPTR     pPartDecodeFunc  /* function to decode partition table */
    )
```

DESCRIPTION To handle a partitioned disk, this function should be called, with *subDev* as the handle returned from **dcacheDevCreate( )**, It is recommended that for efficient operation a single disk cache be allocated for the entire disk and shared by its partitions.

nPart is the maximum number of partitions which are expected for the particular disk drive. Up to 24 (C-Z) partitions per disk are supported.

PARTITION DECODE FUNCTION

An external partition table decode function is provided via the *pPartDecodeFunc* argument, which implements a particular style and format of partition tables, and fill in the results into a table defined as Pn array of **PART_TABLE_ENTRY** types. See **dpartCbio.h** for definition of **PART_TABLE_ENTRY**. The prototype for this function is as follows:

```
    STATUS parDecodeFunc
  (
  CBIO_DEV_ID dev,    /* device from which to read blocks */
  PART_TABLE_ENTRY *pPartTab, /* table where to fill results */
  int nPart       /* # of entries in <pPartTable> */
  )
```

RETURNS **CBIO_DEV_ID** or **NULL** if error creating CBIO device.

ERRNO Not Available

SEE ALSO **dpartCbio**, **dosFsDevCreate( )**.

dpartPartGet( )

NAME

dpartPartGet( ) – retrieve handle for a partition

SYNOPSIS

```
CBIO_DEV_ID dpartPartGet
    (
    CBIO_DEV_ID masterHandle,  /* CBIO handle of the master partition */
    int         partNum        /* partition number from 0 to nPart */
    )
```

DESCRIPTION

This function retrieves a CBIO handle into a particular partition of a partitioned device. This handle is intended to be used with **dosFsDevCreate( )**.

RETURNS

CBIO_DEV_ID or **NULL** if partition is out of range, or *masterHandle* is invalid.

ERRNO

Not Available

SEE ALSO

dpartCbio, **dosFsDevCreate( )**

dshmMuxHwAddrToOff( )

NAME

dshmMuxHwAddrToOff( ) – translate a local address to a shared memory offset

SYNOPSIS

```
uint32_t dshmMuxHwAddrToOff
    (
    const uint_t       hw,
    const void * const addr
    )
```

DESCRIPTION

This routine takes a pointer in shared memory and translates it into a shared memory offset that can be passed via a message to a remote node. This is needed when nodes do not see the shared memory at the same address locally.* It allows then to pass addresses as offset from a common point of reference.

RETURNS

the shared memory offset corresponding to the local address

ERRNO

Not Available

SEE ALSO

dshmMuxLib

dshmMuxHwGet( )

NAME

dshmMuxHwGet( ) – obtain an hardware registration handle based on name

SYNOPSIS

```
int dshmMuxHwGet
    (
    const char * const name
    )
```

DESCRIPTION

This routine returns the handle of a previously registered hardware based on the name used at registration time.

RETURNS

a hardware registration handle, or -1 if it does not exist

ERRNO

Not Available

SEE ALSO

dshmMuxLib

dshmMuxHwLocalAddrGet( )

NAME

dshmMuxHwLocalAddrGet( ) – obtain address of the local node

SYNOPSIS

```
uint16_t dshmMuxHwLocalAddrGet
    (
    const uint_t hw
    )
```

DESCRIPTION

none

RETURNS

the address of the local node, 0x0 to 0xfffe, 0xffff if error

ERRNO

Not Available

SEE ALSO

dshmMuxLib

dshmMuxHwNodesNumGet( )

NAME **dshmMuxHwNodesNumGet( )** – obtain the maximum number of nodes on a hardware bus

SYNOPSIS

```
int dshmMuxHwNodesNumGet
    (
    const uint_t hw
    )
```

DESCRIPTION This routine returns the maximum number of nodes that can exist on a particular hardware bus registered with the MUX.

RETURNS the maximum number of nodes, or -1 if *hw* is invalid

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dshmMuxHwOffToAddr( )

NAME **dshmMuxHwOffToAddr( )** – translate a shared memory offset to a local address

SYNOPSIS

```
void * dshmMuxHwOffToAddr
    (
    const uint_t   hw,
    const uint32_t offset
    )
```

DESCRIPTION This routine takes an offset in shared memory that might have been transmitted via a message and translates it to a local address that can be used as a pointer.

RETURNS the local address corresponding to the shared memory offset

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dshmMuxHwRegister( )

NAME
dshmMuxHwRegister( ) – register a hardware bus with the MUX

SYNOPSIS
```
int dshmMuxHwRegister
    (
    const DSHM_HW_ID           id,        /* bus ID */
    const char * const         name,      /* bus name */
    const uint_t               maxNodes,  /* max number of nodes on the bus */
    const DSHM_HW_HOOKS * const pHooks      /* bus methods (see dshmMuxLib.h)
*/
    )
```

DESCRIPTION
This routine registers a bus supporting DSHM with the MUX, allowing peers to register and thus services to communicate between each other over the bus.

An OS-dependant hardware bus *id* must be provided, as well as a bus *name*, both of which must be unique on the local target. This routine pre- allocates memory for the potential services in the system. Finally, entry points in the bus controller *pHooks* are registered at this point as well.

These hooks include: shared memory allocation and freeing, message transmission, message broadcasting, atomic set (test-and-set, compare-and-swap, read-modify-write), atomic clear (might be left to **NULL** if not needed), shared memory offset to local address translation and vice-versa, retrieval of local node address on the bus, allocation of virtual memory region on the bus, and a fast copy routine.

RETURNS
a hardware registration handle, or -1 if failure

ERRNOS
S_dshm_MUX_HW_NAME_EXISTS
This bus name is already in use.

S_dshm_MUX_HW_TABLE_FULL

and memory allocation failure errnos.

SEE ALSO
dshmMuxLib

dshmMuxHwTasClearGet( )

NAME
dshmMuxHwTasClearGet( ) – obtain the TAS clear routine on this bus

SYNOPSIS
```
DSHM_TAS_CLEAR dshmMuxHwTasClearGet
```

```
    (
    const uint_t hw
    )
```

DESCRIPTION none

RETURNS the atomic set routine, **NULL** if invalid hardware identifier

ERRNO Not Available

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

dshmMuxHwTasGet( )

NAME **dshmMuxHwTasGet( )** – obtain the test-and-set routine on this bus

SYNOPSIS

```
DSHM_TAS dshmMuxHwTasGet
    (
    const uint_t hw
    )
```

DESCRIPTION none

RETURNS the atomic set routine, **NULL** if invalid hardware identifier

ERRNO Not Available

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

dshmMuxLibInit( )

NAME **dshmMuxLibInit( )** – initialize the DSHM MUX

SYNOPSIS

```
void dshmMuxLibInit
    (
    const uint_t maxHwReg,  /* max hw types */
    const uint_t maxSvc     /* max services */
    )
```

DESCRIPTION Prepare the mux to receive registration for hardware buses and services. The maximum amount for each is a configuration parameter. Services cannot register before their hardware bus does a prior registration.

The maximum number of hardware buses *maxHwReg* and services *maxSvc* can be** any positive integer number, but in practice should always be very small numbers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dshmMuxMemAlloc( )

NAME **dshmMuxMemAlloc( )** – allocate shared memory from a specific hardware

SYNOPSIS

```
void * dshmMuxMemAlloc
    (
    const uint_t hw,
    int * const  pSize,
    const int    min
    )
```

DESCRIPTION This routine allocates a specific slab of memory owned by specified hardware *hw*. The allocated slab will be of size **pSize* or closest available if no slab of contiguous memory of greater or equal size is available. Slab is guaranteed to be at least *min* in size. To allow allocation of any size available, a value of zero can be passed to *min*.

NOTE Depending on the hardware, a certain alignment may be forced by the
underlying allocation mechanism.

RETURNS Address of slab if successful or **NULL** otherwise.

ERRNOS Any error from the installed allocation routine for the hardware.

SEE ALSO **dshmMuxLib**

dshmMuxMemFree( )

NAME

dshmMuxMemFree( ) – free allocated shared memory from a specific hardware

SYNOPSIS

```
STATUS dshmMuxMemFree
    (
    const uint_t hw,
    void * const pMem
    )
```

DESCRIPTION

This routine frees a previously allocated slab of memory owned by specified hardware *hw*.

RETURNS

OK if slab and hardware are valid, **ERROR** otherwise.

ERRNOS

Any error from the installed free routine for the hardware.

SEE ALSO

dshmMuxLib

dshmMuxMsgRecv( )

NAME

dshmMuxMsgRecv( ) – receive a message

SYNOPSIS

```
STATUS dshmMuxMsgRecv
    (
    const uint_t hw,       /* hardware on which to receive */
    DSHM(msg),             /* message container */
    int          unused1,  /* possibe future expansion: use 0 */
    int          unused2   /* possibe future expansion: use 0 */
    )
```

DESCRIPTION

This routine processes a message and calls the specified service processing routine previously installed.

This routine should only be called by the hardware receive loop, called either via interrupt or in poll mode.

When this routine calls the service handler, it owns the reference lock on the service object. The service handler is responsible for unlocking the object via a call to **dshmMuxSvcObjRelease( )**.

RETURNS

OK if receive is successful or **ERROR** otherwise.

ERRNOS **S_dshm_MUX_SERVICE_NOT_REGISTERED**

any error from the installed receive routine for the service.

SEE ALSO **dshmMuxLib**

dshmMuxMsgSend( )

NAME **dshmMuxMsgSend( )** – transmit a message

SYNOPSIS

```
STATUS dshmMuxMsgSend
    (
    const uint_t hw,       /* hardware on which to transmit */
    DSHM(msg),             /* message to transmit */
    int          unused1,  /* possibe future expansion: use 0 */
    int          unused2   /* possibe future expansion: use 0 */
    )
```

DESCRIPTION This routine will send a message to a specified destination on a specified hardware based on information provided in the *msg* argument. A broadcast address can be provided.

RETURNS **OK** if send is successful or **ERROR** otherwise.

ERRNOS **S_dshm_MUX_NODE_NOT_REGISTERED**

SEE ALSO **dshmMuxLib**

dshmMuxSvcNodeJoin( )

NAME **dshmMuxSvcNodeJoin( )** – signal services that a node has joined the system

SYNOPSIS

```
void dshmMuxSvcNodeJoin
    (
    const uint_t   hw,   /* on which hw this node sits */
    const uint16_t addr  /* node's unique address on hw */
    )
```

DESCRIPTION This routine is to be called when a node joins the system to allow registered service to perform some action based on the event. This can be something like setting up colleague links, allocating memory for the peer, etc.

This routine should be called by hardware interfaces when they detect that a remote node has appeared on the shared bus.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dshmMuxSvcNodeLeave( )

NAME **dshmMuxSvcNodeLeave( )** – signal services that a node has left the system

SYNOPSIS

```
void dshmMuxSvcNodeLeave
    (
    const uint_t   hw,   /* hardware handle obtained from registration */
    const uint16_t addr  /* node's unique address on the bus */
    )
```

DESCRIPTION This routine is to be called when a node leaves the system to allow registered services to perform some action based on the event. This allows services to update their view of the system and reclaim resources used for interacting with that particular node.

This routine should be called by hardware interfaces when they detect that a remote node has disappeared on the shared bus.

RETURNS N/A

ERRNOS N/A

SEE ALSO **dshmMuxLib**

dshmMuxSvcObjGet( )

NAME **dshmMuxSvcObjGet( )** – retrieve a service object and protect it against deletion

SYNOPSIS

```
void * dshmMuxSvcObjGet
    (
    const uint_t hw,  /* hw on which the service exists */
    const uint_t svc  /* service number, well-known */
    )
```

DESCRIPTION This routine retrieves the reference to the object of a previously registered service *svc* on the hardware bus *hw*. It will also prevent the object from being deleted.

RETURNS A pointer to the object or **NULL** if the service or hardware are not registered.

ERRNOS **S_dshm_MUX_SERVICE_NOT_REGISTERED**

SEE ALSO **dshmMuxLib**

dshmMuxSvcObjRelease( )

NAME **dshmMuxSvcObjRelease( )** – allows modifications to be made on a service object

SYNOPSIS

```
void dshmMuxSvcObjRelease
    (
    const uint_t hw,
    const uint_t svc
    )
```

DESCRIPTION This routine releases a previously obtained reference to the object of a registered service *svc* on the hardware bus *hw*.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dshmMuxSvcRegister( )

NAME **dshmMuxSvcRegister( )** – register a service with the MUX

SYNOPSIS

```
STATUS dshmMuxSvcRegister
    (
    const uint_t              hw,     /* hw on which to register service */
    const uint_t              svc,    /* service number, well-known */
    const void * const        pObj,   /* pointer to service object */
    const DSHM_SVC_HOOKS * const pHooks  /* service receive routine */
    )
```

DESCRIPTION
Register a service with well-known number *svc* with the MUX. A receive hook called when a message arrives on any hardware registered must be provided via the *rx* argument. An argument to the *rx* method can register via *arg* and will be passed to *rx* when it is called.

RETURNS
OK if service if registered successfully or **ERROR** otherwise.

ERRNOS
S_dshm_MUX_SERVICE_TABLE_FULL

SEE ALSO
dshmMuxLib

dshmMuxSvcWithdraw( )

NAME
dshmMuxSvcWithdraw( ) – remove service from MUX

SYNOPSIS
```
STATUS dshmMuxSvcWithdraw
    (
    const uint_t hw,
    const uint_t svc
    )
```

DESCRIPTION
This routine removes the service with well-known name *svc* from the MUX, for the hardware bus *hw*. The **stop** callback registered with the service will be invoked at this point.

After removal, messages intended for it will be discarded by the hardware processing loop.

RETURNS
OK if node withdrawal successful or **ERROR** otherwise.

ERRNOS
S_dshm_MUX_SERVICE_NOT_REGISTERED

SEE ALSO
dshmMuxLib

dshmMuxWidtdrawComplete( )

NAME
dshmMuxWidtdrawComplete( ) – signal service has finished withdrawing

SYNOPSIS
```
STATUS dshmMuxSvcWithdrawComplete
    (
    const uint_t hw,
    const uint_t svc
    )
```

DESCRIPTION To be called by the service when the service determines it can remove itself from the system completely.

WARNING This routine must be called with the reference to the object NOT OWNED by the calling thread. This routine cannot be called from within the **stop** callback registered with the service.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dshmMuxLib**

dsiDataPoolShow( )

NAME **dsiDataPoolShow( )** – display DSI's data pool statistics

SYNOPSIS

```
void dsiDataPoolShow (void)
```

DESCRIPTION This routine displays the statistics for the allocated/available clusters in the DSI data pool. That pool is used to transfer data packets between DSI sockets.

RETURNS N/A

ERRNO Not Available

SEE ALSO **dsiSockLib**, **netPoolShow( )**

dsiSysPoolShow( )

NAME **dsiSysPoolShow( )** – display DSI's system pool statistics

SYNOPSIS

```
void dsiSysPoolShow (void)
```

DESCRIPTION This routine displays the statistics for the allocated/available clusters in the DSI system pool. That pool is used by DSI sockets and their protocols' control blocks.

RETURNS N/A

ERRNO
Not Available

SEE ALSO
dsiSockLib, **netPoolShow( )**

e( )

NAME
e( ) – set or display eventpoints (WindView)

SYNOPSIS
```
STATUS e
    (
    INSTR * addr,           /* where to set eventpoint, or          */
                            /* 0 means display all eventpoints      */
    event_t eventId,        /* event ID                             */
    int     taskNameOrId,   /* task affected; 0 means all tasks     */
    FUNCPTR evtRtn,         /* function to be invoked;              */
                            /* NULL means no function is invoked    */
    int     arg             /* argument to be passed to <evtRtn>    */
    )
```

DESCRIPTION
This routine sets "eventpoints"--that is, breakpoint-like instrumentation markers that can be inserted in code to generate and log an event for use with WindView. Event logging must be enabled with **wvEvtLogEnable( )** for the eventpoint to be logged.

eventId selects the evenpoint number that will be logged: it is in the user event ID range (0-25536).

If *addr* is **NULL**, then all eventpoints and breakpoints are displayed. If *taskNameOrId* is 0, then this event is logged in all tasks. The *evtRtn* routine is called when this eventpoint is hit. If *evtRtn* returns **OK**, then the eventpoint is logged; otherwise, it is ignored. If *evtRtn* is a **NULL** pointer, then the eventpoint is always logged.

Eventpoints are exactly like breakpoints (which are set with the **b( )** command) except in how the system responds when the eventpoint is hit. An eventpoint typically records an event and continues immediately (if *evtRtn* is supplied, this behavior may be different). Eventpoints cannot be used at interrupt level.

To delete an eventpoint, use **bd( )**.

RETURNS
OK, or **ERROR** if *addr* is odd or nonexistent in memory, or if the breakpoint table is full.

ERRNO
N/A

SEE ALSO
dbgLib, **wvEvent( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*

edi( )

NAME

edi( ) – return the contents of register **edi** (also **esi** - **eax**) (x86)

SYNOPSIS

```
int edi
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **edi** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all general registers (**edi** - **eax**): **edi( )** - **eax( )**.

The stack pointer is accessed via **eax( )**.

RETURNS

The contents of register **edi** (or the requested register).

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

edi( )

NAME

edi( ) – return the contents of register **edi** (also **esi** - **eax**) (x86/SimNT)

SYNOPSIS

```
int edi
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **edi** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all address registers (**edi** - **eax**): **edi( )** - **eax( )**.

The stack pointer is accessed via **eax( )**.

RETURNS

The contents of register **edi** (or the requested register).

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

edrBootCountGet( )

NAME **edrBootCountGet( )** – returns the current boot count

SYNOPSIS

```
int edrBootCountGet (void)
```

DESCRIPTION This function returns the number of times the system has been rebooted since ED&R was first initialized.

RETURNS The current boot count, or **ERROR** if it cannot be determined.

ERRNO Not Available

SEE ALSO **edrLib**

edrBootShow( )

NAME **edrBootShow( )** – show all stored boot type ED&R records

SYNOPSIS

```
STATUS edrBootShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of BOOT. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrClear( )

NAME **edrClear( )** – a synonym for edrErrorLogClear

SYNOPSIS

```
STATUS edrClear (void)
```

DESCRIPTION This functions provides command line interface to clear the entire ED&R error log.

RETURNS **OK**, or **ERROR** if there was some problem clearing the log

ERRNO Not Available

SEE ALSO **edrShow**

edrErrLogAttach( )

NAME **edrErrLogAttach( )** – attach to an existing log

SYNOPSIS

```
EDR_ERR_LOG * edrErrLogAttach
    (
    void * pAddr  /* address of an existing log   */
    )
```

DESCRIPTION This routine attaches to a previously created error log, but only if *pAddr* represents a valid existing log.

RETURNS pointer to log, or **NULL** if the pLog doesn't represent a valid log instance.

ERRNO Not Available

SEE ALSO **edrErrLogLib**, **edrErrLogCreate( )**

edrErrLogClear( )

NAME **edrErrLogClear( )** – clear the log's contents

SYNOPSIS

```
BOOL edrErrLogClear
```

```
    (
    EDR_ERR_LOG * pLog  /* pointer to log */
    )
```

DESCRIPTION This routine resets all of the the log's node nodes to the unallocated state.

RETURNS **TRUE**, or **FALSE** if the *pLog* doesn't represent a valid log instance.

ERRNO Not Available

SEE ALSO **edrErrLogLib**, **edrErrLogCreate( )**, **edrErrLogAttach( )**

edrErrLogCreate( )

NAME **edrErrLogCreate( )** – create a new log

SYNOPSIS

```
EDR_ERR_LOG * edrErrLogCreate
    (
    void * pAddr,      /* address to overlay the log at         */
    int    size,       /* length (in bytes) of the log area     */
    int    recordSize  /* length (in bytes) of each record      */
    )
```

DESCRIPTION Creates a new log at address *pAddr* for a length of *size* bytes. Each record in the log is created as size *recordSize*. The size of the record may be no less than **EDR_ERR_LOG_MIN_PAYLOAD_SIZE**. The log has a fixed size overhead of approximately 500 bytes plus an overhead of approximately 150 bytes per record.

If the architecture supports an MMU, the log is write protected after being created.

NOTE The memory provided for the error log must be page aligned and be a multiple of a page size.

RETURNS A pointer to the log, or **NULL** if the log could not be created, or *pAddr* is **NULL**.

ERRNO Not Available

SEE ALSO **edrErrLogLib**, **edrErrLogAttach( )**

edrErrLogIterCreate( )

NAME

edrErrLogIterCreate( ) – create an iterator for traversing the log

SYNOPSIS

```
BOOL edrErrLogIterCreate
    (
    EDR_ERR_LOG *      pLog,   /* pointer to log */
    EDR_ERR_LOG_ITER * pIter,  /* pointer to iter for construction */
    int                start,  /* starting position */
    int                count   /* number of nodes to enumerate */
    )
```

DESCRIPTION

This routine creates an iterator suitable for traversing the set of committed nodes in the log. The starting postion (record number) to begin traversing the log is specified by *start*. The maximum number of records to iterate over is specified by *count*.

RETURNS

TRUE, or **FALSE** if *pLog* points to a corrupt or invalid log.

ERRNO

Not Available

SEE ALSO

edrErrLogLib, **edrErrLogIterNext( )**

edrErrLogIterNext( )

NAME

edrErrLogIterNext( ) – returns the next committed node

SYNOPSIS

```
EDR_ERR_LOG_NODE * edrErrLogIterNext
    (
    EDR_ERR_LOG_ITER * pIter  /* pointer to iterator */
    )
```

DESCRIPTION

This routine returns the next committed node in the log, or **NULL** if there are no more nodes that can be enumerated.

RETURNS

A pointer to next node or **NULL**.

ERRNO

Not Available

SEE ALSO

edrErrLogLib, **edrErrLogIterCreate( )**

edrErrLogMaxNodeCount( )

NAME **edrErrLogMaxNodeCount( )** – return the maximum number of nodes in the log

SYNOPSIS

```
int edrErrLogMaxNodeCount
    (
    EDR_ERR_LOG * pLog  /* pointer to log */
    )
```

DESCRIPTION This routine returns the maximum number of nodes that can be held within the specified log.

RETURNS node count, or **ERROR** if *pLog* is not a valid error log.

ERRNO Not Available

SEE ALSO **edrErrLogLib**

edrErrLogNodeAlloc( )

NAME **edrErrLogNodeAlloc( )** – allocate a node from the error log

SYNOPSIS

```
EDR_ERR_LOG_NODE * edrErrLogNodeAlloc
    (
    EDR_ERR_LOG * pLog  /* pointer to log */
    )
```

DESCRIPTION This routine allocates the next available node from the specified log. Once the node information has been written, the node must be committed to the log using **edrErrLogNodeCommit( )**.

If the architecture supports an MMU, once an node is allocated it is unprotected (ie. writable) until it is commit using **edrErrLogNodeCommit( )**.

RETURNS A node instance, or **NULL** if there are no free nodes.

ERRNO Not Available

SEE ALSO **edrErrLogLib**, **edrErrLogNodeCommit( )**

edrErrLogNodeCommit( )

NAME

edrErrLogNodeCommit( ) – commits a previously allocated node

SYNOPSIS

```
BOOL edrErrLogNodeCommit
    (
    EDR_ERR_LOG *      pLog,  /* pointer to log */
    EDR_ERR_LOG_NODE * pNode  /* pointer to node to commit */
    )
```

DESCRIPTION

This routine commits the previously allocated node *pNode* to the log specified by *pLog*. Once a node becomes committed, it may be re-allocated by **edrErrLogNodeAlloc( )** if necessary.

If the architecture supports an MMU, the node is write protected after being committed.

RETURNS

TRUE if *pNode* is currently allocated, or **FALSE** if *pLog* or *pNode* don't represent valid instances of a log or a node respectively.

ERRNO

Not Available

SEE ALSO

edrErrLogLib, **edrErrLogNodeAlloc( )**

edrErrLogNodeCount( )

NAME

edrErrLogNodeCount( ) – return the number of committed nodes in the log

SYNOPSIS

```
int edrErrLogNodeCount
    (
    EDR_ERR_LOG * pLog  /* pointer to log */
    )
```

DESCRIPTION

This routine returns the number of committed nodes within the specified log.

RETURNS

THe node count, or **ERROR** if *pLog* is not a valid error log.

ERRNO

Not Available

SEE ALSO

edrErrLogLib

edrErrorInject( )

NAME

edrErrorInject( ) – injects an error into the ED&R subsystem

SYNOPSIS

```
STATUS edrErrorInject
    (
    int               kind,        /* severity | facility | option */
    const char *      fileName,    /* name of source file          */
    int               lineNumber,  /* line number of source code   */
    const REG_SET *   pRegSet,     /* current register values      */
    const EXC_INFO *  pExcInfo,    /* CPU-specific exception info  */
    void *            address,     /* faulting address             */
    const char *      msg          /* additional text string       */
    )
```

DESCRIPTION

Warning: This function should not normally be called directly, rather, one of the macros in **edrLib.h** such as **EDR_KERNEL_FATAL_INJECT** should be used instead.

This function takes all the supplied arguments and stores them in an error record, along with numerous other bits of useful information, such as:

- the OS version
- the CPU type and number
- the time at which the error occured
- the current OS context (task, interrupt, exception, RTP)
- a small memory map of the running system
- a code fragment from around the faulting instruction
- a stack trace of the currently active stack

The type of record being injected is represented by the *kind* parameter. The *kind* parameter is a bitwise OR of the following three items:

Severity:

EDR_SEVERITY_FATAL	- a fatal event
EDR_SEVERITY_NONFATAL	- a non-fatal event
EDR_SEVERITY_WARNING	- a warning event
EDR_SEVERITY_INFO	- an information event

Facility:

EDR_FACILITY_KERNEL	- VxWorks kernel events
EDR_FACILITY_INTERRUPT	- interrupt handler events
EDR_FACILITY_INIT	- system startup events
EDR_FACILITY_BOOT	- system boot events
EDR_FACILITY_REBOOT	- system restart events
EDR_FACILITY_RTP	- RTP system events
EDR_FACILITY_USER	- user generated events

Options:

EDR_EXCLUDE_REGISTERS	- don't include registers
EDR_EXCLUDE_TRACEBACK	- don't include stack trace
EDR_EXCLUDE_EXCINFO	- don't include exc info
EDR_EXCLUDE_DISASSEMBLY	- don't include code disssembly
EDR_EXCLUDE_MEMORYMAP	- don't include memory map

From an injection point of view, only the options have an effect on how the record is generated. The severity and facility values are merely stored in the record for subsequent use by the show commands.

If the ED&R subsystem is not yet initialised, then the error-record cannot be written to the log.

LIMITATIONS

Since this function may well be called in an exception handling context, it must be able to run in the limited stack environment for exception handlers. The stack may be as small as a single VM page, i.e. 4K. Thus no large stack-based arrays or other structures should be used by the injection hooks.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK if the error was stored correctly, or **ERROR** if some failure occurs during storage

ERRORS

S_edrLib_NOT_INITIALIZED
if the library was not initialized

S_edrLib_PROTECTION_FAILURE
if the memory could not be protected

SEE ALSO

edrLib

edrErrorInjectHookAdd( )

NAME

edrErrorInjectHookAdd( ) – adds a hook which gets called on error-injection

SYNOPSIS

```
STATUS edrErrorInjectHookAdd
    (
    EDR_ERRINJ_HOOK_FUNCPTR injectHook
    )
```

DESCRIPTION

This function adds a hook to **edrLib** that gets called whenever an error is injected. The hook function (which must be of type **EDR_ERRINJ_HOOK_FUNCPTR**) is invoked with a subset of the parameters passed to **edrErrorInject( )**. The parameters passed to the hook function are as follows:

```
int             kind          /* severity | facility           */
const char *    fileName      /* name of source file           */
int             lineNumber    /* line number of source code    */
void *          address       /* faulting address              */
const char *    msg           /* additional text string        */
```

IMPORTANT NOTE The hook function is called directly from **edrErrorInject( )** and so may be invoked in an interrupt or exception context. Hook functions should therefore make no blocking calls to the VxWorks kernel API, not should they use an excessive amount of stack space.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** or **ERROR** if the hook table is full

ERRORS **S_edrLib_INJECT_HOOK_TABLE_FULL** if the hook table is full.

SEE ALSO **edrLib**, edrErrorInjectHookDelete

edrErrorInjectHookDelete( )

NAME **edrErrorInjectHookDelete( )** – removes an existing error-inject hook

SYNOPSIS

```
STATUS edrErrorInjectHookDelete
    (
    EDR_ERRINJ_HOOK_FUNCPTR injectHook
    )
```

DESCRIPTION This function removes a hook which was added using **edrErrorInjectHookAdd( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** or **ERROR** if the hook was not found in the hook table.

ERRORS **S_edrLib_INJECT_HOOK_NOT_FOUND** if the hook was not found.

SEE ALSO **edrLib**, edrErrorInjectHookAdd

edrErrorInjectPrePostHookAdd( )

NAME

edrErrorInjectPrePostHookAdd( ) – adds a hook which gets called before and after error-injection

SYNOPSIS

```
STATUS edrErrorInjectPrePostHookAdd
    (
    VOIDFUNCPTR hook
    )
```

DESCRIPTION

This function adds a hook to **edrLib** that gets called before and after an error is injected, The hook function (which must be of type FUNCPTR) is invoked with a single integer argument indicating whether it is pre-injection (**EDR_HOOK_TYPE_PRE**), or post-injection (**EDR_HOOK_TYPE_POST**).

These hook points are not generally for use by clients of **edrLib**, but are provided as a means to, for example, kick a hardware watchdog before entering **edrErrorInject( )** in order to ensure that **edrLib** has sufficient time to inject an error-record into the error-log before the watchdog reboots the board.

It should have the following form:-

```
void prePostHook
    (
    int    prePost      /* EDR_HOOK_TYPE_PRE, EDR_HOOK_TYPE_POST */
    );
```

IMPORTANT NOTE

The hook function is called directly from **edrErrorInject( )** and so may be invoked in an interrupt or exception context. Hook functions should therefore make no blocking calls to the VxWorks kernel API, not should they use an excessive amount of stack space.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK or **ERROR** if the hook table is full

ERRORS

S_edrLib_PP_HOOK_TABLE_FULL if the hook table is full.

SEE ALSO

edrLib, edrErrorInjectPrePostHookDelete

edrErrorInjectPrePostHookDelete( )

NAME

edrErrorInjectPrePostHookDelete( ) – removes the existing pre/post hook

SYNOPSIS

```
STATUS edrErrorInjectPrePostHookDelete
    (
    VOIDFUNCPTR hook
    )
```

DESCRIPTION

This function removes a hook which was added using **edrErrorInjectPrePostHookAdd( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK or **ERROR** if the hook was not found in the hook table.

ERRORS

S_edrLib_PP_HOOK_NOT_FOUND if the hook was not found.

SEE ALSO

edrLib, edrErrorInjectPrePostHookAdd

edrErrorInjectTextHookAdd( )

NAME

edrErrorInjectTextHookAdd( ) – adds a hook which gets called on record creation

SYNOPSIS

```
STATUS edrErrorInjectTextHookAdd
    (
    EDR_ERRINJ_TEXT_HOOK_FUNCPTR textHook
    )
```

DESCRIPTION

This function adds a hook to **edrLib** that gets called whenever an error is created. The hook function (which must be of type **EDR_ERRINJ_TEXT_HOOK_FUNCPTR**) is invoked with a pointer and length to a buffer which can be filled with textual information. The parameters passed to the hook function are as follows:

```
char *       p           /* pointer to buffer           */
int          size        /* size of buffer              */
int          kind        /* severity | facility         */
const char * fileName    /* name of source file         */
int          lineNumber  /* line number of source code  */
void *       address     /* faulting address            */
```

The hook must ensure the string written is null terminated. The return value of the hook must be the number of bytes stored, including the trailing null.

IMPORTANT NOTE The hook function is called directly from **edrErrorInject( )** and so may be invoked in an interrupt or exception context. Hook functions should therefore make no blocking calls to the VxWorks kernel API, not should they use an excessive amount of stack space.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** or **ERROR** if the hook table is full

ERRORS **S_edrLib_TEXT_HOOK_TABLE_FULL** if the hook table is full

SEE ALSO **edrLib**, edrErrorInjectTextHookDelete

edrErrorInjectTextHookDelete( )

NAME **edrErrorInjectTextHookDelete( )** – removes the existing text writing hook

SYNOPSIS

```
STATUS edrErrorInjectTextHookDelete
    (
    EDR_ERRINJ_TEXT_HOOK_FUNCPTR textHook
    )
```

DESCRIPTION This function removes a hook which was added using **edrErrorInjectTextHookAdd( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** or **ERROR** if the hook was not found in the hook table.

ERRORS **S_edrLib_TEXT_HOOK_NOT_FOUND** if the hook was not found.

SEE ALSO **edrLib**, edrErrorInjectTextHookAdd

edrErrorLogClear( )

NAME **edrErrorLogClear( )** – clears the ED&R error log

SYNOPSIS

```
STATUS edrErrorLogClear (void)
```

DESCRIPTION This function clears all error records out of the error log. It is destructive (records cannot be undeleted) and should be used with utmost care.

RETURNS **OK** or **ERROR** if the log was not able to be cleared

ERRORS **S_edrLib_NOT_INITIALIZED**
if the library was not initialized

S_edrLib_PROTECTION_FAILURE
if the memory could not be protected

SEE ALSO **edrLib**

edrErrorRecordCount( )

NAME **edrErrorRecordCount( )** – returns the number of error-records in the log

SYNOPSIS
```
int edrErrorRecordCount (void)
```

DESCRIPTION This function returns the total number of ED&R records which are present in the error log.

RETURNS the number of error-records in the log, or **ERROR** if the library has not been initialized

ERRORS **S_edrLib_NOT_INITIALIZED** if the library was not initialized

SEE ALSO **edrLib**

edrErrorRecordDecode( )

NAME **edrErrorRecordDecode( )** – decode one error-record

SYNOPSIS
```
STATUS edrErrorRecordDecode
    (
    EDR_ERROR_RECORD* pER,      /* pointer to error record  */
    char *            pBuf,     /* pointer to output buffer */
    int               bufSize   /* size of output buffer    */
    )
```

DESCRIPTION This routine decodes a single error-record into the provided buffer. If no buffer is provided, the record is decoded to stdout.

RETURNS **OK**, or **ERROR** if the record can't be decoded

ERRNO Not Available

SEE ALSO **edrShow**

edrFatalShow( )

NAME **edrFatalShow( )** – show all stored fatal type ED&R records

SYNOPSIS

```
STATUS edrFatalShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a severity of FATAL. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrFlagsGet( )

NAME **edrFlagsGet( )** – return the ED&R flags which are currently set

SYNOPSIS

```
int edrFlagsGet(void)
```

DESCRIPTION This routine returns all the ED&R flags which have been set to "on". An identical API is provided in the RTP space.

RETURNS an integer with the appropriate bits set

ERRNO Not Available

SEE ALSO **edrSysDbgLib**

edrHelp( )

NAME **edrHelp( )** – prints helpful information on ED&R

SYNOPSIS

```
STATUS edrHelp (void)
```

DESCRIPTION This routine provides the on-line help for the ED&R show commands.

RETURNS **OK**

ERRNO Not Available

SEE ALSO **edrShow**

edrHookShow( )

NAME **edrHookShow( )** – show the list of installed ED&R hook routines

SYNOPSIS

```
STATUS edrHookShow (void)
```

DESCRIPTION This routine shows all the hook routines installed in the various ED&R hook tables, in the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **edrShow**, **edrInjectHookShow( )**, **edrInjectTextHookShow( )**, **edrInjectPrePostHookShow( )**

edrInfoShow( )

NAME **edrInfoShow( )** – show all stored info type ED&R records

SYNOPSIS

```
STATUS edrInfoShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a severity of INFO. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrInitShow( )

NAME **edrInitShow( )** – show all stored init type ED&R records

SYNOPSIS

```
STATUS edrInitShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of INIT. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrInjectHookShow( )

NAME **edrInjectHookShow( )** – show the list of error injection hook routines

SYNOPSIS

```
STATUS edrInjectHookShow (void)
```

DESCRIPTION This routine shows all the error injection routines installed in the ED&R inject hook table, in the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **edrShow**, **edrInjectHookAdd( )**

edrInjectPrePostHookShow( )

NAME **edrInjectPrePostHookShow( )** – show the list of pre/post injection hook routines

SYNOPSIS

```
STATUS edrInjectPrePostHookShow (void)
```

DESCRIPTION This routine shows all the pre/post error injection routines installed in the ED&R inject hook table, in the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **edrShow**, **edrInjectPrePostHookAdd( )**

edrInjectTextHookShow( )

NAME **edrInjectTextHookShow( )** – show the list of text injection hook routines

SYNOPSIS

```
STATUS edrInjectTextHookShow (void)
```

DESCRIPTION This routine shows all the text error injection routines installed in the ED&R inject hook table, in the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **edrShow**, **edrInjectTextHookAdd( )**

edrIntShow( )

NAME **edrIntShow( )** – show all stored interrupt type ED&R records

SYNOPSIS

```
STATUS edrIntShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of INTERRUPT. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrIsDebugMode( )

NAME **edrIsDebugMode( )** – is the ED&R debug mode flag set?

SYNOPSIS

```
BOOL edrIsDebugMode(void)
```

DESCRIPTION This routine returns **TRUE** if the ED&R debug flag is set.

RETURNS **TRUE** if debug flag is set, **FALSE** otherwise.

ERRNO Not Available

SEE ALSO **edrSysDbgLib**

edrKernelShow( )

NAME **edrKernelShow( )** – show all stored kernel type ED&R records

SYNOPSIS

```
STATUS edrKernelShow
```

```
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of KERNEL. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrLibInit( )

NAME **edrLibInit( )** – initializes **edrLib**

SYNOPSIS

```
STATUS edrLibInit
    (
    BOOL isNew,      /* should the PM log area be re-initialized?    */
    int  recordSize  /* size of each ED&R record                     */
    )
```

DESCRIPTION This function initializes the ED&R susbsystem. The parameter *isNew* indicates whether the persistent memory (PM) region used to hold the ED&R records is to be re-initialized. If *isNew* is **FALSE**, **edrLibInit( )** will attempt to open an existing ED&R log PM region and check that it is valid. If the PM region is invalid, an error is returned. Once the PM region is opened successfully, the PM region is write protected. The parameter *recordSize* is used whenever the log is re-initilaised. Its value represents the payload size for each of the records in the log. A value of zero specifies that the built in system default size is to be used.

RETURNS **OK** if the log was successfully created or opened, **ERROR** otherwise.

ERRORS

S_edrLib_PMREGION_ERROR
if the PM region is invalid

S_edrLib_PROTECTION_FAILURE
if the PM region cannot be protected

S_edrLib_ERRLOG_CORRUPTED
if the log appears to be corrupted

S_edrLib_ERRLOG_INCOMPATIBLE
if the log is a newer version

SEE ALSO **edrLib**, usrEdrInit in **usrEdrInit.c**

edrRebootShow( )

NAME **edrRebootShow( )** – show all stored reboot type ED&R records

SYNOPSIS

```
STATUS edrRebootShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of REBOOT. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrRtpShow( )

NAME **edrRtpShow( )** – show all stored rtp type ED&R records

SYNOPSIS

```
STATUS edrRtpShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION This command displays all records stored in the ED&R log which have a facility of RTP. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **edrShow**, **edrShow( )**

edrShow( )

NAME

edrShow( ) – displays the ED&R error log to stdout

SYNOPSIS

```
STATUS edrShow
    (
    int start,     /* starting point */
    int count,     /* number of records to show */
    int facility,  /* limit to specified facility */
    int severity   /* limit to specified severity */
    )
```

DESCRIPTION

This command display all or part of the stored ED&R error log to stdout. The command takes four parameters, the start, count, facility, and severity specifiers.

The *start* parameter specifies the starting record at which the display should begin. If *start* is a positive number, the display begins at *start* records from the beginning of the log. If *start* is a negative number, the display begins at *start* records from the end of the log.

The *count* parameter denotes the number of records to display. A value of zero will display all records.

The *facility* and *severity* parameters limit the display to only those records which match the specified facility and severity. A value of zero will match all facilities and severities.

RETURNS

OK or **ERROR** if the ED&R library was not initialized

ERRNO

S_edrLib_NOT_INITIALIZED

SEE ALSO

edrShow

edrSystemDebugModeGet( )

NAME

edrSystemDebugModeGet( ) – indicates if the system is in **debug** mode

SYNOPSIS

```
BOOL edrSystemDebugModeGet (void)
```

DESCRIPTION

This routine returns the current setting of the system mode debug flag if it has been set, otherwise it assumes it is off.

RETURNS

TRUE if the **debug mode** boot flag is set, **FALSE** if not

ERRNO

Not Available

SEE ALSO **edrSysDbgLib**

edrSystemDebugModeInit( )

NAME **edrSystemDebugModeInit( )** – initialise the system mode debug flag

SYNOPSIS

```
STATUS edrSystemDebugModeInit (void)
```

DESCRIPTION This routine reads the **SYSFLG_SYS_MODE_DEBUG** flag from the boot-flags supplied in the system boot line and set the state of the debug flag.

RETURNS **TRUE** if the **debug mode** boot flag is set, **FALSE** if not

ERRNO Not Available

SEE ALSO **edrSysDbgLib**

edrSystemDebugModeSet( )

NAME **edrSystemDebugModeSet( )** – modifies the system **debug** mode flag

SYNOPSIS

```
void edrSystemDebugModeSet
    (
    BOOL mode
    )
```

DESCRIPTION This routine sets the system debug flag, as maintained by ED&R. It over-rides any setting given in the boot flags.

RETURNS n/a

ERRNO Not Available

SEE ALSO **edrSysDbgLib**

edrUserShow( )

NAME

edrUserShow( ) – show all stored user type ED&R records

SYNOPSIS

```
STATUS edrUserShow
    (
    int start,  /* starting point */
    int count   /* number of records to show */
    )
```

DESCRIPTION

This command displays all records stored in the ED&R log which have a facility of USER. The command accepts a *start* and *count* as documented in **edrShow( )**.

RETURNS

OK or **ERROR**

ERRNO

Not Available

SEE ALSO

edrShow, **edrShow( )**

eflags( )

NAME

eflags( ) – return the contents of the status register (x86)

SYNOPSIS

```
int eflags
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of the status register from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

RETURNS

The contents of the status register.

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

eflags( )

NAME **eflags( )** – return the contents of the status register (x86/SimNT)

SYNOPSIS

```
int eflags
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the status register from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

RETURNS The contents of the status register.

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

elPciRegister( )

NAME **elPciRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void elPciRegister(void)
```

DESCRIPTION This routine registers the elPci driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **tc3c905VxbEnd**

eneRegister( )

NAME **eneRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void eneRegister(void)
```

DESCRIPTION This routine registers the NE2000 driver with VxBus as a child of the PLB bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **ne2000VxbEnd**

envGet( )

NAME **envGet( )** – return a pointer to the environment of a task

SYNOPSIS

```
char ** envGet
    (
    int taskId  /* task for which the environment is to be returned */
    )
```

DESCRIPTION This routine returns a pointer to the environment of a task. If the task has a private environment set, this is what is being referred to then. If *taskId* is **NULL**, then the calling task's environment is retrieved.

RETURNS a pointer to the task's environment, or the pointer to the global environment if the task has none.

ERRNOS **S_objLib_OBJ_ID_ERROR**

SEE ALSO **envLib**

envLibInit( )

NAME **envLibInit( )** – initialize environment variable facility

SYNOPSIS

```
STATUS envLibInit
    (
    BOOL installHooks
    )
```

DESCRIPTION If *installHooks* is **TRUE**, task create and delete hooks are installed that will optionally create and destroy private environments for the task being created or destroyed, depending on the state of **VX_PRIVATE_ENV** in the task options word. If *installHooks* is **FALSE** and a task requires a private environment, it is the application's responsibility to create and destroy the private environment, using **envPrivateCreate( )** and **envPrivateDestroy( )**.

installHooks is controlled by the configuration parameter **ENV_VAR_USE_HOOKS**.

RETURNS **OK**, or **ERROR** if an environment cannot be allocated or the hooks cannot be installed.

ERRNOS N/A

SEE ALSO **envLib**

envPrivateCreate( )

NAME **envPrivateCreate( )** – create a private environment

SYNOPSIS

```
STATUS envPrivateCreate
    (
    int taskId,     /* task to have private environment */
    int envSource   /* -1 = make an empty private environment */
                    /* 0 = copy global env to new private env */
                    /* task id = copy the specified task's env */
    )
```

DESCRIPTION This routine creates a private set of environment variables for a specified task, if the environment variable task create hook is not installed.

Based on the *envSource* argument, the environment is created in one of three ways:

envSource	Copy Behavior
---------	------------------------------------
-1	create an empty environment for *taskId*
0	copy global environment to *taskId*'s new private environment
a task id	Given a task ID, copy the task's private environment for *taskId*

NOTE This API does not protect against the tasks from deletion while the environment information is being copied from one task to another. The user should take care not to delete the *taskId* nor the *envSource* task id while this routine is being used.

RETURNS **OK**, or **ERROR** if memory is insufficient or source environment is **NULL**.

ERRNOS
S_objLib_OBJ_ID_ERROR
taskId is invalid.

SEE ALSO **envLib**, **envLibInit( )**, **envPrivateDestroy( )**

envPrivateDestroy( )

NAME **envPrivateDestroy( )** – destroy a private environment

SYNOPSIS

```
STATUS envPrivateDestroy
    (
    int taskId  /* task with private env to destroy */
    )
```

DESCRIPTION This routine destroys a private set of environment variables that were created with **envPrivateCreate( )**. Calling this routine is unnecessary if the environment variable task create hook is installed and the task was spawned with **VX_PRIVATE_ENV**.

RETURNS **OK**, or **ERROR** if the task does not exist.

ERRNOS **S_objLib_OBJ_ID_ERROR**
taskId is invalid.

SEE ALSO **envLib**, **envPrivateCreate( )**

envShow( )

NAME **envShow( )** – display the environment for a task

SYNOPSIS

```
void envShow
    (
    int taskId  /* task for which environment is printed */
    )
```

DESCRIPTION This routine prints to standard output all the environment variables for a specified task or the global environment. If *taskId* is **NULL**, then the calling task's environment is displayed.

RETURNS N/A

ERRNOS **S_objLib_OBJ_ID_ERROR**

SEE ALSO **envLib**

errnoGet( )

NAME **errnoGet( )** – get the error status value of the calling task

SYNOPSIS

```
int errnoGet (void)
```

DESCRIPTION This routine gets the current error status value. It is provided for compatibility with previous versions of VxWorks.

RETURNS The current error status value.

ERRNO N/A

SEE ALSO **errnoLib**, **errnoSet( )**, **errnoOfTaskGet( )**

errnoOfTaskGet( )

NAME **errnoOfTaskGet( )** – get the error status value of a specified task

SYNOPSIS

```
int errnoOfTaskGet
    (
    int taskId  /* task ID, 0 means current task */
    )
```

DESCRIPTION This routine gets the error status most recently set in the TCB of a specified task. If *taskId* is zero, the calling task is assumed.

This routine is provided primarily for debugging purposes. Normally, tasks access **errno** directly to set and get their own error status values.

RETURNS The error status of the specified task, or **ERROR** if the task does not exist.

ERRNO N/A

SEE ALSO **errnoLib**, **errnoSet( )**, **errnoGet( )**

errnoOfTaskSet( )

NAME

errnoOfTaskSet( ) – set the error status value of a specified task

SYNOPSIS

```
STATUS errnoOfTaskSet
    (
    int taskId,     /* task ID, 0 means current task */
    int errorValue  /* error status value */
    )
```

DESCRIPTION

This routine sets the error status value in the TCB for a specified task. If *taskId* is zero, the calling task is assumed.

This routine is provided primarily for debugging purposes. Normally, tasks access **errno** directly to set and get their own error status values.

RETURNS

OK, or **ERROR** if the task does not exist.

ERRNO

N/A

SEE ALSO

errnoLib, **errnoSet( )**, **errnoOfTaskGet( )**

errnoSet( )

NAME

errnoSet( ) – set the error status value of the calling task

SYNOPSIS

```
STATUS errnoSet
    (
    int errorValue  /* error status value to set */
    )
```

DESCRIPTION

This routine sets the current **errno** with a specified error status. It is provided for compatibility with previous versions of VxWorks.

RETURNS

OK, or **ERROR** if the interrupt nest level is too deep.

ERRNO

N/A

SEE ALSO

errnoLib, **errnoGet( )**, **errnoOfTaskSet( )**

etsecRegister( )

NAME **etsecRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void etsecRegister(void)
```

DESCRIPTION This routine registers the ETSEC driver with VxBus as a child of the PLB bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **vxbEtsecEnd**

eventClear( )

NAME **eventClear( )** – Clear the calling task's events register

SYNOPSIS

```
STATUS eventClear (void)
```

DESCRIPTION This routine clears the calling task's events register. Since events can be received at any time, the caller cannot assume its events register is actually cleared by the time this routine returns unless interrupts are locked when this routine is called.

RETURNS **OK** on success or **ERROR**

ERRNO **S_intLib_NOT_ISR_CALLABLE**
Routine was called from an ISR.

SEE ALSO **eventLib**, **eventReceive( )**

eventReceive( )

NAME **eventReceive( )** – Wait for event(s)

SYNOPSIS

```
STATUS eventReceive
    (
    UINT32 events,           /* events task is waiting to occur */
    UINT8  options,           /* user options */
    int    timeout,           /* ticks to wait */
    UINT32 *pEventsReceived   /* events occurred are returned through this */
    )
```

DESCRIPTION

Pends calling task until one or all wanted *events* have been received. When the specified events have been received, they are copied from the events register to the variable pointed to by *pEventsReceived*, and the events register is cleared (by default).

The *options* parameter is used to control various aspects of this routine's behaviour. One of which is to specify if the caller wishes to wait for all events to be received or only one. One of the following must be specified:

EVENTS_WAIT_ANY (0x1)
Wait for any one of the wanted events.

EVENTS_WAIT_ALL (0x0)
Wait for all wanted events.

Another option is to specify if the events written to *pEventsReceived* are only those received and wanted or all events received. Note that an event can be received at any time, including before **eventReceive( )** is called. By default this routine returns only wanted events unless the following option is specified:

EVENTS_RETURN_ALL (0x2)
Causes the routine to return all received events whether they are wanted (as specified in *events*) or not. It also causes all events to be cleared from the task's events register.

The third option available allows the caller to specify if the received unwanted events are to be cleared from the calling task's events register. They are cleared by default unless the following option is specified:

EVENTS_KEEP_UNWANTED (0x4)
Tells this routine not to clear unwanted events. In cases where the **EVENTS_RETURN_ALL** option is used, all events are cleared even if this option is selected. Wanted events are always cleared hence this option has not effect on them.

Lastly, it is possible to retrieve events that have already been received without affecting the events register by selecting the following option:

EVENTS_FETCH (0x80)
If this option is specified, the contents of the calling task's events register are copied to the location pointed to by *pEventsReceived* and the routine returns immediately. The events are not cleared from the register. The *events* and *timeout* arguments are ignored and so are all other options specified.

The *timeout* parameter specifies the number of ticks to wait for wanted events. It can also have the following special values:

NO_WAIT (0)
Return immediately, even if no events have arrived.

WAIT_FOREVER (-1)
Never time out.

The received events are copied to the location pointed to by *pEventsReceived* even when the routine returns **ERROR** unless a **NULL** pointer is passed.

WARNING This routine may not be used from interrupt level because ISRs do not have events registers.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK** on success or **ERROR**

ERRNO

S_eventLib_TIMEOUT
Wanted events not received before specified timeout expired.

S_eventLib_NOT_ALL_EVENTS
Wanted events not received at the time of the call. This error can only occur if **NO_WAIT** is specified in *timeout*

S_objLib_OBJ_DELETED
Task is waiting for events from a resource that has been destroyed. See **semEvLib** and **msgQEvLib** documentation for more information.

S_intLib_NOT_ISR_CALLABLE
Function was called from ISR.

S_eventLib_ZERO_EVENTS
The *events* parameter has been passed a value of 0.

SEE ALSO **eventLib**, **semEvLib**, **msgQEvLib**, **eventSend( )**

eventSend( )

NAME **eventSend( )** – Send event(s)

SYNOPSIS

```
STATUS eventSend
    (
    int    taskId,  /* task events will be sent to */
    UINT32 events   /* events to send              */
    )
```

DESCRIPTION Sends specified *events* to a task. Passing a taskId of **NULL** causes the calling task to send events to itself. This routine can be used by an ISR to send events to a task.

Because an event is actually a bit in the 32 bit word *events*, the sending process consists of bitwise ORing *events* with the present contents of the destination task's events register. Therefore the process is said to be non-destructive since the events that may already be present in the task's events register are not affected.

Sending an event to a task that already has the event in its events register does not alter the contents of the register.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK** on success or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
taskId is invalid.

S_eventLib_NULL_TASKID_AT_INT_LEVEL
Routine was called from ISR with a taskId of **NULL**.

SEE ALSO **eventLib**, **eventReceive( )**

excConnect( )

NAME **excConnect( )** – connect a C routine to an exception vector (PowerPC)

SYNOPSIS

```
STATUS excConnect
    (
    VOIDFUNCPTR * vector,  /* exception vector to attach to */
    VOIDFUNCPTR   routine  /* routine to be called */
    )
```

DESCRIPTION

This routine connects a specified C routine to a specified exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and calls the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) placed on the stack of the task that has produced the exception. The structure of the ESF used to save the registers is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the registers values. The prototype of this C routine is described below:

```
    void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS

OK, always.

ERRNO

Not Available

SEE ALSO

excArchLib, **excIntConnect( )**, **excVecSet( )**

---

excCrtConnect( )

NAME

excCrtConnect( ) – connect a C routine to a critical exception vector (PowerPC 403)

SYNOPSIS

```
STATUS excCrtConnect
    (
    VOIDFUNCPTR * vector,  /* exception vector to attach to */
    VOIDFUNCPTR   routine  /* routine to be called */
    )
```

DESCRIPTION

This routine connects a specified C routine to a specified critical exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and call the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) which is placed on the stack of the task that has produced the exception. The ESF structure is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the register values. The prototype of this C routine is as follows:

```
    void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS

OK, always.

ERRNO

Not Available

SEE ALSO **excArchLib**, **excIntConnect( )**, excIntCrtConnect, **excVecSet( )**

excHookAdd( )

NAME **excHookAdd( )** – specify a routine to be called with exceptions

SYNOPSIS

```
void excHookAdd
    (
    FUNCPTR excepHook  /* routine to call when exceptions occur */
    )
```

DESCRIPTION This routine specifies a routine that will be called when hardware exceptions occur. The specified routine is called after normal exception handling, which includes displaying information about the error. Upon return from the specified routine, the task that incurred the error is suspended.

The exception handling routine should be declared as:

```
    void myHandler
        (
        int      task,    /* ID of offending task              */
        int      vecNum,  /* exception vector number           */
        <ESFxx>  *pEsf    /* pointer to exception stack frame */
        )
```

where *task* is the ID of the task that was running when the exception occurred. *ESFxx* is architecture-specific and can be found by examining **/target/h/arch/***arch*/esf*arch*.h; for example, the PowerPC uses ESFPPC.

This facility is normally used by **dbgLib( )** to activate its exception handling mechanism. If an application provides its own exception handler, it will supersede the **dbgLib** mechanism.

RETURNS N/A

ERRNOS N/A

SEE ALSO **excLib**

excInit( )

NAME **excInit( )** – initialize the exception handling package

SYNOPSIS

```
STATUS excInit
    (
    UINT maxIsrJobs  /* must be a power of two */
    )
```

DESCRIPTION This routine initializes the interrupt-level job deferral facility. This facility provides the ability to defer function execution to task level from interrupt level.

RETURNS **OK**, or **ERROR** if the tExcTask cannot be spawned.

ERRNOS N/A

SEE ALSO **excLib**

excIntConnect( )

NAME **excIntConnect( )** – connect a C routine to an asynchronous exception vector (PowerPC, ARM)

SYNOPSIS

```
STATUS excIntConnect
    (
    VOIDFUNCPTR * vector,  /* exception vector to attach to */
    VOIDFUNCPTR   routine  /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous exception vector.

When the C routine is invoked, interrupts are still locked. It is the responsibility of the C routine to re-enable the interrupt.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

NOTE On PowerPC, the vector is typically the external interrupt vector 0x500 and the decrementer vector 0x900. An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines.

Before saving the requested registers, the interrupt stub switches from the current task stack to the interrupt stack. For nested interrupts, no stack-switching is performed, because the interrupt is already set.

NOTE On the ARM, the address of *routine* is stored in a function pointer to be called by the stub installed on the IRQ exception vector following an asynchronous exception. This routine is responsible for determining the interrupt source and despatching the correct handler for that source.

Before calling the routine, the interrupt stub switches to SVC mode, changes to a separate interrupt stack and saves necessary registers. In the case of a nested interrupt, no SVC stack switch occurs.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **excArchLib**, **excConnect( )**, **excVecSet( )**

excIntCrtConnect( )

NAME **excIntCrtConnect( )** – connect a C routine to a critical interrupt vector (PowerPC 403)

SYNOPSIS

```
STATUS excIntCrtConnect
    (
    VOIDFUNCPTR * vector,  /* exception vector to attach to */
    VOIDFUNCPTR   routine  /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous critical exception vector such as the critical external interrupt vector (0x100), or the watchdog timer vector (0x1020). An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs, the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines.

When the C routine is invoked, interrupts are still locked. It is the C routine's responsibility to re-enable interrupts.

The routine can be any normal C routine, except that it must not invoke certain operating system functions that may block or perform I/O operations.

Before the requested registers are saved, the interrupt stub switches from the current task stack to the interrupt stack. In the case of nested interrupts, no stack switching is performed, because the interrupt stack is already set.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **excArchLib**, **excConnect( )**, excCrtConnect, **excVecSet( )**

excJobAdd( )

NAME **excJobAdd( )** – request a task-level function call from interrupt level

SYNOPSIS

```
STATUS excJobAdd
    (
    VOIDFUNCPTR func,
    int         arg1,
    int         arg2,
    int         arg3,
    int         arg4,
    int         arg5,
    int         arg6
    )
```

DESCRIPTION This routine allows interrupt level code to request a function call to be executed by the tExcTask at task-level.

WARNING Care must be taken when pushing jobs to tExcTask. Jobs that may block, hang, or generate exceptions must be avoided since blocking or suspension of the tExcTask may cause other parts of the system to misbehave.

RETURNS **OK**. Otherwise **ERROR** if job posting fails.

ERRNO N/A

SEE ALSO **excLib**

excVecGet( )

NAME **excVecGet( )** – get a CPU exception vector (PowerPC, ARM)

SYNOPSIS

```
FUNCPTR excVecGet
```

```
    (
    FUNCPTR * vector  /* vector offset */
    )
```

DESCRIPTION This routine returns the address of the C routine currently connected to *vector*.

RETURNS The address of the C routine.

ERRNO Not Available

SEE ALSO **excArchLib**, **excVecSet( )**

excVecInit( )

NAME **excVecInit( )** – initialize the exception/interrupt vectors

SYNOPSIS

```
STATUS excVecInit (void)
```

DESCRIPTION This routine sets all exception vectors to point to the appropriate default exception handlers. These handlers will safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

MC680x0:
All vectors from vector 2 (address 0x0008) to 255 (address 0x03fc) are initialized. Vectors 0 and 1 contain the reset stack pointer and program counter.

MIPS:
All MIPS exception, trap, and interrupt vectors are set to default handlers.

x86:
All vectors from vector 0 (offset (0x0000) to 255 (offset 0x07f8) are initialized to default handlers. A global variable excDoBell controls the bell that takes 660 microsecs in the default exception show routine. The default value is **TRUE**. To turn the bell off, set it **FALSE**.

PowerPC:
There are 48 vectors and only vectors that are used are initialized.

SH:
There are 256 vectors, initialized with the default exception handler (for exceptions) or the unitialized interrupt handler (for interupts). On SH-2, vectors 0 and 1 contain the power-on reset program counter and stack pointer. Vectors 2 and 3 contain the manual reset program counter and stack pointer. On SH-3 and SH-4 processors the vector table is located at (vbr + 0x800), and the (exception code / 8) value is used as vector offset.

The first two vectors are reserved for special use: "trapa #0" (offset 0x0) to implement software breakpoint, and "trapa #1' (offset 0x4) to detect integer zero divide exception.

ARM:
All exception vectors are initialized to default handlers except 0x14 (Address) which is now reserved on the ARM and 0x1C (FIQ), which is not used by VxWorks.

SimSolaris/SimNT:
This routine does nothing on both simulators and always returns **OK**.

NOTE This routine is usually called from the system start-up routine, **usrInit( )**, in **usrConfig.c**. It must be called before interrupts are enabled.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **excArchLib**, **excLib**

excVecSet( )

NAME **excVecSet( )** – set a CPU exception vector (PowerPC, ARM)

SYNOPSIS

```
void excVecSet
    (
    FUNCPTR * vector,   /* vector offset */
    FUNCPTR   function  /* address to place in vector */
    )
```

DESCRIPTION This routine specifies the C routine that will be called when the exception corresponding to *vector* occurs. This routine does not create the exception stub; it simply replaces the C routine to be called in the exception stub.

NOTE ARM On the ARM, there is no **excConnect( )** routine, unlike the PowerPC. The C routine is attached to a default stub using **excVecSet( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **excArchLib**, **excVecGet( )**, **excConnect( )**, **excIntConnect( )**

exit( )

NAME

exit( ) – exit a task (ANSI)

SYNOPSIS

```
void exit
    (
    int code  /* code stored in TCB for delete hooks */
    )
```

DESCRIPTION

This routine is called by a task to cease to exist as a task. It is called implicitly when the "main" routine of a spawned task is exited. The *code* parameter will be stored in the **WIND_TCB** for possible use by the delete hooks, or post-mortem debugging.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

taskLib, **taskDelete( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output (***stdlib.h**), The VxWorks Programmer's Guide

expf( )

NAME

expf( ) – compute an exponential value (ANSI)

SYNOPSIS

```
float expf
    (
    float x  /* exponent */
    )
```

DESCRIPTION

This routine returns the exponential of *x* in single precision.

RETURNS

The single-precision exponential value of *x*.

ERRNO

Not Available

SEE ALSO

mathALib

fabsf( )

NAME

fabsf( ) – compute an absolute value (ANSI)

SYNOPSIS

```
float fabsf
    (
    float v  /* number to return the absolute value of */
    )
```

DESCRIPTION

This routine returns the absolute value of *v* in single precision.

RETURNS

The single-precision absolute value of *v*.

ERRNO

Not Available

SEE ALSO

mathALib

fastStrSearch( )

NAME

fastStrSearch( ) – Search by optimally choosing the search algorithm

SYNOPSIS

```
char * fastStrSearch
    (
    char * pattern,       /* pattern to search for */
    int    patternLen,    /* length of the pattern */
    char * buffer,        /* text buffer to search in */
    int    bufferLen,     /* length of the text buffer */
    BOOL   caseSensitive  /* case-sensitive search? */
    )
```

DESCRIPTION

Depending on the pattern size, this function uses either the Boyer-Moore-Sunday algorithm or the Brute Force algorithm. The Boyer-Moore-Sunday algorithm requires pre-processing, therefore for small patterns it is better to use the Brute Force algorithm.

RETURNS

A pointer to the located pattern, or a **NULL** pointer if the pattern is not found

ERRNO

Not Available

SEE ALSO

strSearchLib

fccRegister( )

NAME

fccRegister( ) – register with the VxBus subsystem

SYNOPSIS

```
void fccRegister(void)
```

DESCRIPTION

This routine registers the FCC driver with VxBus as a child of the PCI bus type.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

fccVxbEnd

fchmod( )

NAME

fchmod( ) – change the permission mode of a file

SYNOPSIS

```
int fchmod
    (
    int    fd,
    mode_t mode
    )
```

DESCRIPTION

The fchmod function changes or assigns the mode of a file. The mode of a file specifies its permissions and other attributes. Note that this routine receives **open file descriptor** as the first argument compairing to chmod routine.

The value of *mode* is bitwise inclusive OR of the permissions to be assigned

These permission constants are defined in *sys/stat.h* as follows:

S_IRUSR
    Read permission, owner.

S_IWUSR
    Write permission, owner.

S_IXUSR
    Execute/search permission, owner.

S_IRWXU
    Read/write/execute permission, owner.

S_IRGRP
    Read permission, group.

S_IWGRP
Write permission, group.

S_IXGRP
Execute/search permission, group.

S_IRWXG
Read/write/execute permission, group.

S_IROTH
Read permission, other.

S_IWOTH
Write permission, other.

S_IXOTH
Execute/search permission, other.

S_IRWXO
Read/write/execute permission, other.

RETURNS

If it succeeds, returns **OK**, 0. Otherwise, **ERROR**, -1 is returned, errno is set to indicate the error and no change is done to the file.

The following example changes the mode of the file "myFile" to owner Read/write/execute, group Read and other Read:

```
fd = open ("myFile",  O_RDONLY, 0 );
status = fchmod (fd, S_IRWXU | S_IRGRP | S_IROTH );
```

ERRNO

EBADF
The *fd* argument is not a valid open file.

others
Other errors reported by device driver.

SEE ALSO

fsPxLib

fcntl( )

NAME

fcntl( ) – perform control functions over open files

SYNOPSIS

```
int fcntl
   (
   int fd,
   int command,
   ...
   )
```

DESCRIPTION

The **fcntl( )** function provides for control over open files. The *fd* argument is an open file descriptor. The **fcntl( )** function may take a third argument whose data type, value and use depend upon the value of *command* which specifies the operation to be performed by **fcntl( )**.

RETURNS

Not Available

ERRNO

EMFILE
Ran out of file descriptors

EBADF
Bad file descriptor number.

ENOSYS
Device driver does not support the ioctl command.

ENXIO
Device and its driver are removed. **close( )** should be called to release this file descriptor.

Other
Other errors reported by device driver.

SEE ALSO

ioLib

fdatasync( )

NAME

fdatasync( ) – synchronize a file data

SYNOPSIS

```
int fdatasync
    (
    int fd  /* file descriptor of the file to datasync */
    )
```

DESCRIPTION

The function forces all currently queued I/O operations associated with the file indicated by *fd* to the synchronized I/O completion state.

The functionality is as described for **fsync( )** with the exception that all I/O operations are completed as defined for synchronised I/O data integrity completion.

RETURNS

Upon successful completion, **OK**, 0 is returned. Otherwise, **ERROR**, -1 returned and errno is set to indicate the error. If the **fdatasync( )** function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRNO

SEE ALSO **fsPxLib**, **fsync( )**

fdprintf( )

NAME **fdprintf( )** – write a formatted string to a file descriptor

SYNOPSIS

```
int fdprintf
    (
    int          fd,    /* file descriptor to write to */
    const char * fmt,   /* format string to write */
    ...                 /* optional arguments to format */
    )
```

DESCRIPTION This routine writes a formatted string to a specified file descriptor. Its function and syntax are otherwise identical to **printf( )**.

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

ERRNO Not Available

SEE ALSO **fioLib**, **printf( )**

fecRegister( )

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

SYNOPSIS

```
void fecRegister(void)
```

DESCRIPTION This routine registers the FEC driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **fecVxbEnd**

feiRegister( )

NAME
feiRegister( ) – register with the VxBus subsystem

SYNOPSIS
```
void feiRegister(void)
```

DESCRIPTION
This routine registers the Intel 8255x driver with VxBus as a child of the PCI bus type.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
fei8255xVxbEnd

ffsLsb( )

NAME
ffsLsb( ) – find least significant bit set

SYNOPSIS
```
int ffsLsb
    (
    UINT32 i  /* value in which to find first set bit */
    )
```

DESCRIPTION
This routine finds the least significant bit set in the 32 bit argument passed to it and returns the index of that bit. Bits are numbered starting at 1 from the least signifficant bit. A return value of zero indicates that the value passed is zero.

RETURNS
index of least significant bit set, or zero

ERRNO
N/A

SEE ALSO
ffsLib

ffsMsb( )

NAME
ffsMsb( ) – find most significant bit set

SYNOPSIS
```
int ffsMsb
```

```
    (
    UINT32 i  /* value in which to find first set bit */
    )
```

DESCRIPTION This routine finds the most significant bit set in the 32 bit argument passed to it and returns the index of that bit. Bits are numbered starting at 1 from the least signifficant bit. A return value of zero indicates that the value passed is zero.

RETURNS index of most significant bit set, or zero

ERRNO N/A

SEE ALSO **ffsLib**

fileUploadPathClose( )

NAME **fileUploadPathClose( )** – close the event-destination file

SYNOPSIS

```
void wvFileUploadPathClose
    (
    UPLOAD_ID pathId  /* generic upload-path descriptor */
    )
```

DESCRIPTION This routine closes the file associated with *pathId* that is serving as a destination for event data.

RETURNS N/A

ERRNO Not Available

SEE ALSO **wvFileUploadPathLib**, **wvFileUploadPathCreate( )**

fioBaseLibInit( )

NAME **fioBaseLibInit( )** – initialize the formatted I/O support library

SYNOPSIS

```
void fioBaseLibInit (void)
```

DESCRIPTION This routine initializes the formatted I/O support library. It should be called once in **usrRoot( )** when formatted I/O functions such as **printf( )** and **scanf( )** are used.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fioBaseLib**

fioFormatV( )

NAME **fioFormatV( )** – convert a format string

SYNOPSIS

```
int fioFormatV
    (
    FAST const char *fmt,        /* format string */
    va_list         vaList,      /* pointer to varargs list */
    FUNCPTR         outRoutine,  /* handler for args as they're formatted */
    int             outarg       /* argument to routine */
    )
```

DESCRIPTION This routine is used by the **printf( )** family of routines to handle the actual conversion of a format string. The first argument is a format string, as described in the entry for **printf( )**. The second argument is a variable argument list *vaList* that was previously established.

As the format string is processed, the result will be passed to the output routine whose address is passed as the third parameter, *outRoutine*. This output routine may output the result to a device, or put it in a buffer. In addition to the buffer and length to output, the fourth argument, *outarg*, will be passed through as the third parameter to the output routine. This parameter could be a file descriptor, a buffer address, or any other value that can be passed in an "int".

The output routine should be declared as follows:

```
    STATUS outRoutine
        (
        char *buffer, /* buffer passed to routine            */
        int  nchars,  /* length of buffer                    */
        int  outarg   /* arbitrary arg passed to fmt routine */
        )
```

The output routine should return **OK** if successful, or **ERROR** if unsuccessful.

RETURNS The number of characters output, or **ERROR** if the output routine returned **ERROR**.

ERRNO Not Available

SEE ALSO **fioBaseLib**

fioLibInit( )

NAME **fioLibInit( )** – initialize the formatted I/O support library

SYNOPSIS

```
void fioLibInit (void)
```

DESCRIPTION This routine initializes the formatted I/O support library. It should be called once in **usrRoot( )** when formatted I/O functions such as **printf( )** and **scanf( )** are used.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fioLib**

fioRdString( )

NAME **fioRdString( )** – read a string from a file

SYNOPSIS

```
int fioRdString
    (
    int       fd,          /* fd of device to read */
    FAST char string[],    /* buffer to receive input */
    int       maxbytes     /* max no. of chars to read */
    )
```

DESCRIPTION This routine puts a line of input into *string*. The specified input file descriptor is read until *maxbytes*, an **EOF**, an EOS, or a newline character is reached. A newline character or **EOF** is replaced with EOS, unless *maxbytes* characters have been read.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The length of the string read, including the terminating EOS; or **EOF** if a read error occurred or end-of-file occurred without reading any other character.

ERRNO Not Available

SEE ALSO **fioLib**

fioRead( )

NAME

fioRead( ) – read a buffer

SYNOPSIS

```
int fioRead
    (
    int    fd,       /* file descriptor of file to read */
    char * buffer,   /* buffer to receive input */
    int    maxbytes  /* maximum number of bytes to read */
    )
```

DESCRIPTION

This routine repeatedly calls the routine **read( )** until *maxbytes* have been read into *buffer*. If **EOF** is reached, the number of bytes read will be less than *maxbytes*.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

The number of bytes read, or **ERROR** if there is an error during the read operation.

ERRNO

Not Available

SEE ALSO

fioLib, **read( )**

floorf( )

NAME

floorf( ) – compute the largest integer less than or equal to a specified value (ANSI)

SYNOPSIS

```
float floorf
    (
    float v  /* value to find the floor of */
    )
```

DESCRIPTION

This routine returns the largest integer less than or equal to *v*, in single precision.

RETURNS

The largest integral value less than or equal to *v*, in single precision.

ERRNO

Not Available

SEE ALSO

mathALib

fmodf( )

NAME

fmodf( ) – compute the remainder of x/y (ANSI)

SYNOPSIS

```
float fmodf
    (
    float x,  /* numerator   */
    float y   /* denominator */
    )
```

DESCRIPTION

This routine returns the remainder of *x*/*y* with the sign of *x*, in single precision.

RETURNS

The single-precision modulus of *x*/*y*.

ERRNO

Not Available

SEE ALSO

mathALib

formatTrans( )

NAME

formatTrans( ) – Format a transaction disk.

SYNOPSIS

```
STATUS formatTrans
    (
    int fd,        /* rawFS file descriptor */
    int blkshift,  /* blk size shifter */
    int overhead,  /* scaled overhead */
    int type       /* type flag */
    )
```

DESCRIPTION

The *fd* argument should be the result of opening a rawFS disk. (Because we just use ordinary read/write/fstat calls, this can also be any ordinary file descriptor for a file on a disk; we will just pretend that that file is a whole disk. But in general it should be a rawFS.)

The *blkshift* argument is used to increase the logical block size of the underlying device. This reduces the TRFS overhead, but increases the minimum size of any file on the dosFS atop this TRFS layer. That is, the TRFS overhead goes down but the file system overhead goes up. In general this should just be 0.

The *overhead* parameter is 10 times the percent of the disk itself to use as TRFS transactional space, i.e., an argument of 50 gives 5%, while an argument of 100 gives 10%. A value of 0 results in the default of 5%.

The *type* parameter should be one of:

- **FORMAT_REGULAR** (0): puts transaction master records (TMRs
  at the beginning and end of the volume.
- **FORMAT_TFFS** (1): moves the first TMR to sector 1, leaving
  sector 0 available for other purposes (such as TFFS internals).
- **FORMAT_DOS** (2): not actually supported (yet?).

RETURNS **OK** if all went well, **ERROR** otherwise.

ERRNO **EINVAL** – invalid arguments or inappropriate underlying device E2BIG – underlying device too big other errno set by rawFS

SEE ALSO **xbdTrans**, **usrFormatTrans( )**

fpathconf( )

NAME **fpathconf( )** – determine the current value of a configurable limit

SYNOPSIS

```
long fpathconf
    (
    int fd,   /* file descriptor of the file */
    int name  /* Value to query */
    )
```

DESCRIPTION The **fpathconf( )** and **pathconf( )** functions provide a method for the application to determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

RETURNS The current value is returned if valid with the query. Otherwise, **ERROR**, -1 returned and errno may be set to indicate the error. There are many reasons to return **ERROR**. If the variable corresponding to name has no limit for the path or file descriptor, both **pathconf( )** and **fpathconf( )** return -1 without changing errno.

ERRNO

SEE ALSO **fsPxLib**, **pathconf( )**

fppInit( )

NAME

fppInit( ) – initialize floating-point coprocessor support

SYNOPSIS

```
void fppInit (void)
```

DESCRIPTION

This routine initializes floating-point coprocessor support and must be called before using the floating-point coprocessor. This is done automatically by the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_HW_FP** is defined.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

fppLib

fppProbe( )

NAME

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

SYNOPSIS

```
STATUS fppProbe (void)
```

DESCRIPTION

This routine determines whether there is a floating-point coprocessor in the system.

The implementation of this routine is architecture-dependent:

MC680x0, x86, SH-4:

This routine sets the illegal coprocessor opcode trap vector and executes a coprocessor instruction. If the instruction causes an exception, **fppProbe( )** returns **ERROR**. Note that this routine saves and restores the illegal coprocessor opcode trap vector that was there prior to this call.

The probe is only performed the first time this routine is called. The result is stored in a static and returned on subsequent calls without actually probing.

MIPS:

This routine simply reads the R-Series status register and reports the bit that indicates whether coprocessor 1 is usable. This bit must be correctly initialized in the BSP.

ARM:

This routine currently returns **ERROR** to indicate no floating-point coprocessor support.

SimNT, **SimSolaris**:
This routine currently returns **OK**.

RETURNS **OK**, or **ERROR** if there is no floating-point coprocessor.

ERRNO Not Available

SEE ALSO **fppArchLib**

fppRestore( )

NAME **fppRestore( )** – restore the floating-point coprocessor context

SYNOPSIS

```
void fppRestore
    (
    FP_CONTEXT * pFpContext  /* where to restore context from */
    )
```

DESCRIPTION This routine restores the floating-point coprocessor context. The context restored is:

&**MC680x0**:
- registers **fpcr**, **fpsr**, and **fpiar**
- registers **f0 - f7**
- internal state frame (if **NULL**, the other registers are not saved.)

&**MIPS**:
- register **fpcsr**
- registers **fp0 - fp31**

&**SH-4**:
- registers **fpcsr** and **fpul**
- registers **fr0 - fr15**
- registers **xf0 - xf15**

&**x86**:
108 byte old context with fsave and frstor instruction
- control word, status word, tag word,
- instruction pointer,
- instruction pointer selector,
- last FP instruction op code,
- data pointer,
- data pointer selector,

- registers **st/mm0** - **st/mm7** (10 bytes * 8)
512 byte new context with fxsave and fxrstor instruction
- control word, status word, tag word,
- last FP instruction op code,
- instruction pointer,
- instruction pointer selector,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)
- registers **xmm0** - **xmm7** (16 bytes * 8)

&**ARM**:
- currently, on this architecture, this routine does nothing.

&**SimSolaris**:
- register **fsr**
- registers **f0** - **f31**

&**SimNT**:
- this routine does nothing on Windows simulator.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fppArchLib**, **fppSave( )**

fppSave( )

NAME **fppSave( )** – save the floating-point coprocessor context

SYNOPSIS

```
void fppSave
    (
    FP_CONTEXT * pFpContext  /* where to save context */
    )
```

DESCRIPTION This routine saves the floating-point coprocessor context. The context saved is:

&**MC680x0**:
- registers **fpcr**, **fpsr**, and **fpiar**
- registers **f0** - **f7**

- internal state frame (if **NULL**, the other registers are not saved.)

&**MIPS**:
- register **fpcsr**
- registers **fp0** - **fp31**

&**SH-4**:
- registers **fpcsr** and **fpul**
- registers **fr0** - **fr15**
- registers **xf0** - **xf15**

&**x86**:
108 byte old context with fsave and frstor instruction
- control word, status word, tag word,
- instruction pointer,
- instruction pointer selector,
- last FP instruction op code,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)

512 byte new context with fxsave and fxrstor instruction
- control word, status word, tag word,
- last FP instruction op code,
- instruction pointer,
- instruction pointer selector,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)
- registers **xmm0** - **xmm7** (16 bytes * 8)

&**ARM**:
- currently, on this architecture, this routine does nothing.

&**SimSolaris**:
- register **fsr**
- registers **f0** - **f31**

&**SimNT**:
- this routine does nothing on Windows simulator. Floating point registers are saved by Windows.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fppArchLib**, **fppRestore( )**

fppShowInit( )

NAME **fppShowInit( )** – initialize the floating-point show facility

SYNOPSIS

```
void fppShowInit (void)
```

DESCRIPTION This routine links the floating-point show facility into the VxWorks system. It 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**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fppShow**

fppTaskRegsGet( )

NAME **fppTaskRegsGet( )** – Gets FPU context for a task

SYNOPSIS

```
STATUS   fppTaskRegsGet
    (
    int        task,
    FPREG_SET *pFpRegSet
    )
```

DESCRIPTION Gets the FPU context for a task.

RETURNS
OK on success.
ERROR otherwise

ERRNO
S_coprocLib_INVALID_OPERATION
S_coprocLib_INVALID_ARGUMENT
S_coprocLib_NO_COPROC_SUPPORT

SEE ALSO
aimFppLib

fppTaskRegsGet( )

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

SYNOPSIS
```
STATUS fppTaskRegsGet
    (
    int         task,      /* task to get info about */
    FPREG_SET * pFpRegSet  /* ptr to floating-point register set */
    )
```

DESCRIPTION
This routine copies a task's floating-point registers and/or status registers to the locations whose pointers are passed as parameters. The floating-point registers are copied into an array containing all the registers.

NOTE
This routine only works well if *task* is not the calling task. If a task tries to discover its own registers, the values will be stale (that is, left over from the last task switch).

RETURNS
OK, or **ERROR** if there is no floating-point support or there is an invalid state.

ERRNO
Not Available

SEE ALSO
fppArchLib, **fppTaskRegsSet( )**

fppTaskRegsSet( )

NAME
fppTaskRegsSet( ) – Sets FPU context for a task

SYNOPSIS
```
STATUS   fppTaskRegsSet
    (
    int       task,
    FPREG_SET *pFpRegSet
    )
```

DESCRIPTION Sets the FPU context for a task.

RETURNS **OK** on success.
ERROR otherwise

ERRNO **S_coprocLib_INVALID_OPERATION**
S_coprocLib_INVALID_ARGUMENT
S_coprocLib_NO_COPROC_SUPPORT

SEE ALSO **aimFppLib**

fppTaskRegsSet( )

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

SYNOPSIS

```
STATUS fppTaskRegsSet
    (
    int         task,      /* task to set registers for */
    FPREG_SET * pFpRegSet  /* ptr to floating-point register set */
    )
```

DESCRIPTION This routine loads the specified values into the TCB of a specified task. The register values are copied from the array at *pFpRegSet*.

RETURNS **OK**, or **ERROR** if there is no floating-point support or there is an invalid state.

ERRNO Not Available

SEE ALSO **fppArchLib**, **fppTaskRegsGet( )**

fppTaskRegsShow( )

NAME **fppTaskRegsShow( )** – print the contents of a task's floating-point registers

SYNOPSIS

```
void fppTaskRegsShow
```

```
     (
     int task  /* task to display floating point registers for */
     )
```

DESCRIPTION This routine prints to standard output the contents of a task's floating-point registers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **fppShow**

free( )

NAME **free( )** – free a block of memory from the system memory partition (ANSI)

SYNOPSIS

```
void free
     (
     void * ptr  /* pointer to block of memory to free */
     )
```

DESCRIPTION This routine returns to the free memory pool (kernel heap) a block of memory previously allocated with **malloc( )**, **calloc( )**, **memalign( )**, **realloc( )** or **valloc( )**. If *ptr* is a null pointer, no action occurs.

RETURNS N/A

ERRNO Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

SEE ALSO **memPartLib**, **malloc( )**, **calloc( )**, **memPartFree( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: General Utilities* (**stdlib.h**)

fsEventUtilInit( )

NAME **fsEventUtilInit( )** – Initialize the file system event utlility library

SYNOPSIS

```
STATUS fsEventUtilLibInit
```

```
    (
    void
    )
```

DESCRIPTION none

RETURNS **OK** on success, **ERROR** on failure

ERRNO Not Available

SEE ALSO **fsEventUtilLib**

fsMonitorInit( )

NAME **fsMonitorInit( )** – Initialize the fsMonitor

SYNOPSIS

```
STATUS fsMonitorInit(void)
```

DESCRIPTION This routine initializes the fsMonitor library.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **fsMonitor**

fsPathAddedEventRaise( )

NAME **fsPathAddedEventRaise( )** – Raise a "path added" event

SYNOPSIS

```
void fsPathAddedEventRaise
    (
    char * coreIOPath
    )
```

DESCRIPTION This routine raises an event with the event reporting framework when the specified path has been added to core I/O by a file system. This routine will cause the wait for path handler(s) to run.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **fsEventUtilLib**

fsPathAddedEventSetup( )

NAME **fsPathAddedEventSetup( )** – Setup to wait for a path

SYNOPSIS

```
STATUS fsPathAddedEventSetup
    (
    FS_PATH_WAIT_STRUCT * pWaitData,
    char *                path
    )
```

DESCRIPTION This routine registers with the ERF to wait for the specified path to be added to core I/O by a file system. This is mainly used my file system formatters when the eject the current file system and wait for rawFs to instantiate.

RETURNS **OK** on success or **ERROR** on error.

ERRNO Not Available

SEE ALSO **fsEventUtilLib**

fsWaitForPath( )

NAME **fsWaitForPath( )** – wait for a path

SYNOPSIS

```
STATUS fsWaitForPath
    (
    FS_PATH_WAIT_STRUCT *pWaitData
    )
```

DESCRIPTION This routine waits for a path to be added to core I/O. The function, fsPathAddedEventSetup, must be prior for pWaitData to be setup. This function simply waits on the semaphore provided in pWaitData which will be given by the wait for path handler.

RETURNS **OK** on success or **ERROR** on failure

ERRNO

Not Available

SEE ALSO

fsEventUtilLib

fsmGetDriver( )

NAME

fsmGetDriver( ) – Get the XBD name of a mapping based on the path

SYNOPSIS

```
STATUS fsmGetDriver
    (
    char      *volume,  /* core I/O pathname */
    devname_t driver    /* xbd driver name */
    )
```

DESCRIPTION

This routine gets the XBD name which is currently mapped to volume, if such a mapping exists.

The *volume* parameter specifies the pathname for which a driver name is to be retrieved.

The *driver* parameter is the resultant of the name mapping, if found.

RETURNS

OK if the mapping is found, or **ERROR** if no such mapping exists.

ERRNO

Not Available

SEE ALSO

fsMonitor

fsmGetVolume( )

NAME

fsmGetVolume( ) – get the pathname based on an XBD name mapping

SYNOPSIS

```
STATUS fsmGetVolume
    (
    char      *driver,  /* XBD driver name */
    fsmName_t volume    /* core I/O pathname */
    )
```

DESCRIPTION

This routine retrieves the pathname associated with an XBD name if such a mapping exists.

The *driver* parameter specifies a driver name for which a pathname is to be retrieved.

The *volume* parameter specifies the resultant pathname.

RETURNS **OK** if the mapping is retrieved, or **ERROR** if it does not exist.

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmNameInstall( )

NAME **fsmNameInstall( )** – Add a mapping between an XBD name and a pathname

SYNOPSIS

```
STATUS fsmNameInstall
    (
    char *driver,  /* XBD driver name */
    char *volume   /* core I/O pathname */
    )
```

DESCRIPTION This routine creates a mapping between the driver name and the pathname specified by volume. This mapping will persist until removed by fsmNameUninstall.

The *driver* parameter specifies the XBD name to be mapped.

The *volume* parameter specifies the Core I/O path that driver is to be mapped into.

RETURNS **OK** if successful or **ERROR** if the name cannot be added

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmNameMap( )

NAME **fsmNameMap( )** – map an XBD name to a Core I/O path

SYNOPSIS

```
STATUS fsmNameMap
    (
    devname_t xbdName,  /* XBD name */
    fsmName_t volName   /* core I/O path */
    )
```

DESCRIPTION

This function maps an XBD name to a path in Core I/O, either by an explicit mapping specified by **fsmNameInstall( )** or by using the XBD name to create a Core I/O pathname. This function always succeeds.

The *xbdName* parameter specifies the name of the device to be mapped.

The *volName* parameter specifies the resultant mapped name.

RETURNS

OK

ERRNO

Not Available

SEE ALSO

fsMonitor

fsmNameUninstall( )

NAME

fsmNameUninstall( ) – remove an XBD name to pathname mapping

SYNOPSIS

```
STATUS fsmNameUninstall
    (
    char *driver  /* driver name */
    )
```

DESCRIPTION

This routine removes a name mapping added by fsmNameInstall. After invocation of this routine, the F/S monitor will create a pathname based on the XBD name instead of using a name mapping.

The *driver* parameter specifies the name of the XBD for which a mapping is to be removed. All occurences of driver are removed.

RETURNS

OK if the name mapping is removed or **ERROR** if the mapping is not found.

ERRNO

Not Available

SEE ALSO

fsMonitor

fsmProbeInstall( )

NAME

fsmProbeInstall( ) – install F/S probe and instantiator functions

SYNOPSIS

```
STATUS fsmProbeInstall
```

```
    (
    fsmProbeFunc probe,  /* probe routine to install */
    fsmInstFunc  inst    /* instantiator routine to install */
    )
```

DESCRIPTION This routine installs the file system probe and instantiator functions. When a new file system is discovered, the file system monitor will call these functions to test for, and instantiate a particular file system type. If either function fails, then the file system is not created.

The *probe* parameter specifies the probe function to be used.

The *inst* parameter specifies the instantiator function to be used when the probe function succeeds.

RETURNS **OK** on success, or **ERROR** if an error is detected.

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmProbeUninstall( )

NAME **fsmProbeUninstall( )** – remove a file system probe

SYNOPSIS

```
STATUS fsmProbeUninstall
    (
    fsmProbeFunc probe  /* probe routine to uninstall */
    )
```

DESCRIPTION This routine removes all probe-instantiator pairs that match the probe parameter.

The *probe* parameter specifies the probe function of the probe-instantiator pair to be removed.

RETURNS 0 if a probe is removed or **ERROR** if the probe is not found

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmUnmountHookAdd( )

NAME **fsmUnmountHookAdd( )** – Add an unmount hook function

SYNOPSIS

```
STATUS fsmUnmountHookAdd
    (
    FUNCPTR fn
    )
```

DESCRIPTION This routine adds a hook routine to run when a vnode based file system unmounts.

The *fn* parameter specifies the hook function.

RETURNS **OK** if there is space in the table and *fn* is not **NULL**

/NOMANUAL

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmUnmountHookDelete( )

NAME **fsmUnmountHookDelete( )** – Remove an unmount hook function

SYNOPSIS

```
STATUS fsmUnmountHookDelete
    (
    FUNCPTR fn
    )
```

DESCRIPTION This routine removes a hook routine to run when a vnode based file system unmounts.

The *fn* parameter specifies the hook function.

RETURNS **OK** if *fn* is found in the table. **ERROR** otherwise

/NOMANUAL

ERRNO Not Available

SEE ALSO **fsMonitor**

fsmUnmountHookRun( )

NAME

fsmUnmountHookRun( ) – Runs the unmount hook functions

SYNOPSIS

```
void fsmUnmountHookRun
    (
    DEV_HDR *pDev
    )
```

DESCRIPTION

This routine is called my the vnode layer when a file system unmounts to run the unmount hook functions.

The *pDev* parameter is passed to each hook function

/NOMANUAL

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

fsMonitor

fstat( )

NAME

fstat( ) – get file status information (POSIX)

SYNOPSIS

```
STATUS fstat
    (
    int         fd,     /* file descriptor for file to check */
    struct stat *pStat  /* pointer to stat structure */
    )
```

DESCRIPTION

This routine obtains various characteristics of a file (or directory). The file must already have been opened using **open( )** or **creat( )**. The *fd* parameter is the file descriptor returned by **open( )** or **creat( )**.

The *pStat* parameter is a pointer to a **stat** structure (defined in **stat.h**). This structure must be allocated before **fstat( )** is called.

Upon return, the fields in the **stat** structure are updated to reflect the characteristics of the file.

RETURNS

OK or **ERROR**, the result of the **ioctl( )** command to the filesystem driver.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

Other
Other errors reported by device driver.

SEE ALSO

dirLib, **stat( )**, **ls( )**

fstatfs( )

NAME

fstatfs( ) – get file status information (POSIX)

SYNOPSIS

```
STATUS fstatfs
    (
    int           fd,     /* file descriptor for file to check */
    struct statfs *pStat  /* pointer to statfs structure */
    )
```

DESCRIPTION

This routine obtains various characteristics of a file system. A file in the file system must already have been opened using **open( )** or **creat( )**. The *fd* parameter is the file descriptor returned by **open( )** or **creat( )**.

The *pStat* parameter is a pointer to a **statfs** structure (defined in **stat.h**). This structure must be allocated before **fstat( )** is called.

Upon return, the fields in the **statfs** structure are updated to reflect the characteristics of the file system. Note that for DosFS, the fields **f_files** and **f_ffree** are meaningless and are set to -1.

RETURNS

OK or **ERROR**, from the **ioctl( )** command.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

Other
Other errors reported by device driver.

SEE ALSO

dirLib, **statfs( )**, **ls( )**

fsync( )

NAME

fsync( ) – synchronize a file

SYNOPSIS

```
int fsync
    (
    int fd  /* file descriptor of the file to sync */
    )
```

DESCRIPTION

This function moves all modified data and attributes of the file descriptor *fd* to a storage device. When **fsync( )** returns, all in-memory modified copies of buffers associated with *fd* have been written to the physical medium. It forces all outstanding data operations to synchronized file integrity completion.

RETURNS

Upon successful completion, **OK**, 0 is returned. Otherwise, **ERROR**, -1 returned and errno is set to indicate the error. If the **fsync( )** function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRNO

SEE ALSO

fsPxLib, **fdatasync( )**

ftruncate( )

NAME

ftruncate( ) – truncate a file (POSIX)

SYNOPSIS

```
int ftruncate
    (
    int   fildes,  /* fd of file to truncate */
    off_t length   /* length to truncate file */
    )
```

DESCRIPTION

This routine truncates a file to a specified size.

RETURNS

0 (**OK**) or -1 (**ERROR**) if unable to truncate file.

ERRNO

EROFS
File resides on a read-only file system.

EBADF
File is open for reading only.

EINVAL
File descriptor refers to a file on which this operation is impossible.

SEE ALSO **ftruncate**

g0( )

NAME **g0( )** – return the contents of register g0 (also g1-g7) (SimSolaris)

SYNOPSIS

```
int g0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of global register g0 from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all global registers (g0 - g7): **g0( )** - **g7( )**.

RETURNS The contents of register g0 (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

geiRegister( )

NAME **geiRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void geiRegister(void)
```

DESCRIPTION This routine registers the gei driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **gei825xxVxbEnd**

getOptServ( )

NAME **getOptServ( )** – parse parameter string into argc, argv format

SYNOPSIS
```
STATUS getOptServ
    (
    char * ParamString,
    const char * progName,      /* program name value for argv[0] */
    int * argc,
    char * argvloc[],
    int argvlen
    )
```

DESCRIPTION none

RETURNS **OK** if all arguments were successfully stored; otherwise, **ERROR**.

ERRNO Not Available

SEE ALSO **getopt**

getenv( )

NAME **getenv( )** – get an environment variable (ANSI)

SYNOPSIS
```
char *getenv
    (
    FAST const char *name  /* env variable to get value for */
    )
```

DESCRIPTION This routine searches the environment list (see the UNIX BSD 4.3 manual entry for **environ(5V)**) for a string of the form "name=value" and returns the value portion of the string, if the string is present; otherwise it returns a **NULL** pointer.

RETURNS A pointer to the string value, or a **NULL** pointer.

ERRNOS N/A

SEE ALSO **envLib**, **envLibInit( )**, **putenv( )**, 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**)

getopt( )

NAME

getopt( ) – parse argc/argv argument vector (POSIX)

SYNOPSIS

```
int getopt
    (
    int          nargc,
    char * const *nargv,
    const char   *ostr
    )
```

DESCRIPTION

Decodes arguments passed in an argc/argv[] vector

The parameters nargc and nargv are the argument count and argument array as passed to **main( )**. The argument ostr is a string of recognized option characters; if a character is followed by a colon, the option takes an argument.

The variable optind is the index of the next element of the nargv[] vector to be processed. It shall be initialized to 1 by the system, and **getopt( )** shall update it when it finishes with each element of nargv[]. When an element of nargv[] contains multiple option characters, it is unspecified how **getopt( )** determines which options have already been processed.

The **getopt( )** function shall return the next option character (if one is found) from nargv that matches a character in ostr, if there is one that matches. If the option takes an argument, **getopt( )** shall set the variable optarg to point to the option-argument as follows:

If the option was the last character in the string pointed to by an element of nargv, then optarg shall contain the next element of nargv, and optind shall be incremented by 2. If the resulting value of optind is greater than nargc, this indicates a missing option-argument, and **getopt( )** shall return an error indication.

Otherwise, optarg shall point to the string following the option character in that element of nargv, and optind shall be incremented by 1.

If, when **getopt( )** is called:

nargv[optind] is a null pointer nargv[optind] is not the character - nargv[optind] points to the string "-"

getopt( ) shall return -1 without changing optind. If:

nargv[optind] points to the string "--"

getopt( ) shall return -1 after incrementing optind.

If **getopt( )** encounters an option character that is not contained in ostr, it shall return the question-mark (**?**) character. If it detects a missing option-argument, it shall return the colon character (**:**) if the first character of ostr was a colon, or a question-mark character (**?**) otherwise. In either case, **getopt( )** shall set the variable optopt to the option character that caused the error. If the application has not set the variable opterr to 0 and the first character of ostr is not a colon, **getopt( )** shall also print a diagnostic message to stderr in the format specified for the getopts utility.

The **getopt( )** function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURNS

The **getopt( )** function shall return the next option character specified on the command line.

A colon (**:**) shall be returned if **getopt( )** detects a missing argument and the first character of ostr was a colon (**:**).

A question mark (**?**) shall be returned if **getopt( )** encounters an option character not in ostr or detects a missing argument and the first character of ostr was not a colon (**:**).

Otherwise, **getopt( )** shall return -1 when all command line options are parsed.

ERRNO

Not Available

SEE ALSO

getopt, POSIX

getoptInit( )

NAME

getoptInit( ) – initialize the getopt state structure

SYNOPSIS

```
void getoptInit
    (
    GETOPT_ID pArg  /* Pointer to getopt structure to be initialized */
    )
```

DESCRIPTION

This function initializes the structure, *pGetOpt* that is used to maintain the getopt state. This structure is passed to **getopt_r( )** which is a reentrant threadsafe version of the standard **getopt( )** call. This function must be called before calling **getopt_r( )**

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

getopt

getopt_r( )

NAME

getopt_r( ) – parse argc/argv argument vector (POSIX)

SYNOPSIS

```
int getopt_r
    (
    int          nargc,
    char * const *nargv,
    const char   *ostr,
    GETOPT_ID    pGetOpt
    )
```

DESCRIPTION

This function is a reentrant version of the **getopt( )** function. The non-reentrant version keeps the getopt state in global variables across multiple calls made by the application, while this reentrant version keeps the state in the structure provided by the caller, thus allowing multiple callers to use getopt simultaneously without requiring any synchronization between them.

The parameters *nargc* and *nargv* are the argument count and argument array as passed to **main( )**. The argument *ostr* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument. The argument *pGetOpt* points to the structure allocated by the caller to keep track of the getopt state. Prior to calling getopt_r, it is the caller responsibility to initialize this structure by calling **getoptInit( )**.

The variable pGetOpt->optind is the index of the next element of the nargv[] vector to be processed. **getopt_r( )** shall update it when it finishes with each element of nargv[]. When an element of nargv[] contains multiple option characters, it is unspecified how **getopt_r( )** determines which options have already been processed.

The **getopt_r( )** function shall return the next option character (if one is found) from nargv that matches a character in ostr, if there is one that matches. If the option takes an argument, **getopt_r( )** shall set the variable pGetOpt->optarg to point to the option-argument as follows:

If the option was the last character in the string pointed to by an element of nargv, then pGetOpt->optarg shall contain the next element of nargv, and pGetOpt->optind shall be incremented by 2. If the resulting value of pGetOpt->optind is greater than nargc, this indicates a missing option-argument, and **getopt_r( )** shall return an error indication.

Otherwise, pGetOpt->optarg shall point to the string following the option character in that element of nargv, and pGetOpt->optind shall be incremented by 1.

If, when **getopt_r( )** is called:

nargv[pGetOpt->optind] is a null pointer nargv[pGetOpt->optind] is not the character - nargv[pGetOpt->optind] points to the string "-"

getopt_r( ) shall return -1 without changing pGetOpt->optind. If:

nargv[pGetOpt->optind]   points to the string "--"

getopt_r( ) shall return -1 after incrementing pGetOpt->optind.

If **getopt_r( )** encounters an option character that is not contained in ostr, it shall return the question-mark (**?**) character. If it detects a missing option-argument, it shall return the colon character (**:**) if the first character of ostr was a colon, or a question-mark character (**?**) otherwise. In either case, **getopt_r( )** shall set the variable pGetOpt->optopt to the option character that caused the error. If the application has not set the variable pGetOpt->opterr to 0 and the first character of ostr is not a colon, **getopt_r( )** shall also print a diagnostic message to stderr in the format specified for the getopts utility.

This function is reentrant and thread-safe.

RETURNS The **getopt_r( )** function shall return the next option character specified on the command line.

A colon (**:**) shall be returned if **getopt_r( )** detects a missing argument and the first character of ostr was a colon (**:**).

A question mark (**?**) shall be returned if **getopt_r( )** encounters an option character not in ostr or detects a missing argument and the first character of ostr was not a colon (**:**).

Otherwise, **getopt_r( )** shall return -1 when all command line options are parsed.

ERRNO Not Available

SEE ALSO **getopt**, POSIX

h( )

NAME **h( )** – display or set the size of shell history

SYNOPSIS

```
void h
    (
    int size  /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This command displays or sets the size of VxWorks shell history. If no argument is specified, shell history is displayed. If *size* is specified, that number of the most recent commands is saved for display. The value of *size* is initially 20.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **shellHistory( )**, **ledLib**, the VxWorks programmer guides.

hashFuncIterScale( )

NAME

hashFuncIterScale( ) – iterative scaling hashing function for strings

SYNOPSIS

```
int hashFuncIterScale
    (
    int           elements,  /* number of elements in hash table */
    H_NODE_STRING *pHNode,   /* pointer to string keyed hash node */
    int           seed       /* seed to be used as scalar */
    )
```

DESCRIPTION

This hashing function interprets the key as a pointer to a null terminated string. A seed of 13 or 27 appears to work well. It calculates the hash as follows:

```
 for (tkey = pHNode->string; *tkey != '\0'; tkey++)
hash = hash * seed + (unsigned int) *tkey;

 hash &= (elements - 1);
```

RETURNS

integer between 0 and (elements - 1)

ERRNO

N/A

SEE ALSO

hashLib

hashFuncModulo( )

NAME

hashFuncModulo( ) – hashing function using remainder technique

SYNOPSIS

```
int hashFuncModulo
    (
    int        elements,  /* number of elements in hash table */
    H_NODE_INT *pHNode,   /* pointer to integer keyed hash node */
    int        divisor    /* divisor */
    )
```

DESCRIPTION

This hashing function interprets the key as a 32 bit quantity and applies the standard hashing function: h (k) = K mod D, where D is the passed divisor. The result of the hash function is masked to the appropriate number of bits to ensure the hash is not greater than (elements - 1).

RETURNS

integer between 0 and (elements - 1)

ERRNO N/A

SEE ALSO **hashLib**

hashFuncMultiply( )

NAME **hashFuncMultiply( )** – multiplicative hashing function

SYNOPSIS

```
int hashFuncMultiply
    (
    int        elements,   /* number of elements in hash table */
    H_NODE_INT *pHNode,    /* pointer to integer keyed hash node */
    int        multiplier  /* multiplier */
    )
```

DESCRIPTION This hashing function interprets the key as a unsigned integer quantity and applies the standard hashing function: h (k) = leading N bits of (B * K), where N is the appropriate number of bits such that the hash is not greater than (elements - 1). The overflow of B * K is discarded. The value of B is passed as an argument. The choice of B is similar to that of the seed to a linear congruential random number generator. Namely, B's value should take on a large number (roughly 9 digits base 10) and end in ...x21 where x is an even number. (Don't ask... it involves statistics mambo jumbo)

RETURNS integer between 0 and (elements - 1)

ERRNO N/A

SEE ALSO **hashLib**

hashKeyCmp( )

NAME **hashKeyCmp( )** – compare keys as 32 bit identifiers

SYNOPSIS

```
BOOL hashKeyCmp
    (
    H_NODE_INT *pMatchHNode,  /* hash node to match */
    H_NODE_INT *pHNode,       /* hash node in table to compare to */
    int        keyCmpArg      /* argument ignored */
    )
```

DESCRIPTION

This routine compares hash node keys as 32 bit identifiers. The argument keyCmpArg is unneeded by this comparator.

RETURNS

TRUE if keys match or, **FALSE** if keys do not match.

ERRNO

N/A

SEE ALSO

hashLib

hashKeyStrCmp( )

NAME

hashKeyStrCmp( ) – compare keys based on strings they point to

SYNOPSIS

```
BOOL hashKeyStrCmp
    (
    H_NODE_STRING *pMatchHNode,  /* hash node to match */
    H_NODE_STRING *pHNode,       /* hash node in table to compare to */
    int            keyCmpArg     /* argument ignored */
    )
```

DESCRIPTION

This routine compares keys based on the strings they point to. The strings must be null terminated. The routine **strcmp( )** is used to compare keys. The argument keyCmpArg is unneeded by this comparator.

RETURNS

TRUE if keys match or, **FALSE** if keys do not match.

ERRNO

N/A

SEE ALSO

hashLib

hashTblCreate( )

NAME

hashTblCreate( ) – create a hash table

SYNOPSIS

```
HASH_ID hashTblCreate
    (
    int     sizeLog2,   /* number of elements in hash table log 2 */
    FUNCPTR keyCmpRtn,  /* function to test keys for equivalence */
    FUNCPTR keyRtn,     /* hashing function to generate hash from key */
    int     keyArg      /* argument to hashing function */
    )
```

DESCRIPTION

This routine creates a hash table 2^sizeLog2 number of elements. The hash table is carved from the caller's heap via malloc (2). To accommodate the list structures associated with the table, the actual amount of memory allocated will roughly eight times the number of elements requested. Additionally, two routines must be specified to dictate the behavior of the hashing table. The first routine, keyCmpRtn, is the key comparator function and the second routine, keyRtn, is the hashing function.

The hashing function's role is to disperse the hash nodes added to the table as evenly throughout the table as possible. The hashing function receives as its parameters the number of elements in the table, a pointer to the **HASH_NODE** structure, and finally the keyArg parameter passed to this routine. The keyArg may be used to seed the hashing function. The hash function returns an index between 0 and (elements - 1). Standard hashing functions are available in this library.

The keyCmpRtn parameter specifies the other function required by the hash table. This routine tests for equivalence of two **HASH_NODES**. It returns a boolean, **TRUE** if the keys match, and **FALSE** if they differ. As an example, a hash node may contain a **HASH_NODE** followed by a key which is an unsigned integer identifiers, or a pointer to a string, depending on the application. Standard hash node comparators are available in this library.

RETURNS

HASH_ID, or **NULL** if hash table could not be created.

ERRNO

Possible errnos generated by this routine include:

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory large enough to satisfy the allocation request.

SEE ALSO

hashLib, **hashFuncIterScale( )**, **hashFuncModulo( )**, **hashFuncMultiply( )**, **hashKeyCmp( )**, **hashKeyStrCmp( )**

hashTblDelete( )

NAME

hashTblDelete( ) – delete a hash table

SYNOPSIS

```
STATUS hashTblDelete
    (
    HASH_ID hashId  /* id of hash table to delete */
    )
```

DESCRIPTION

This routine deletes the specified hash table and frees the associated memory. The hash table is marked as invalid.

RETURNS

OK, or **ERROR** if *hashId* is invalid.

ERRNO Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

SEE ALSO **hashLib**, **hashTblDestroy( )**, **hashTblTerminate( )**

hashTblDestroy( )

NAME **hashTblDestroy( )** – destroy a hash table

SYNOPSIS

```
STATUS hashTblDestroy
    (
    HASH_ID hashId,  /* id of hash table to destroy */
    BOOL    dealloc  /* deallocate associated memory */
    )
```

DESCRIPTION This routine destroys the specified hash table and optionally frees the associated memory. The hash table is marked as invalid.

RETURNS **OK**, or **ERROR** if *hashId* is invalid.

ERRNO Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

SEE ALSO **hashLib**, **hashTblDelete( )**, **hashTblTerminate( )**

hashTblEach( )

NAME **hashTblEach( )** – call a routine for each node in a hash table

SYNOPSIS

```
HASH_NODE *hashTblEach
    (
    HASH_ID hashId,      /* hash table to call routine for */
    FUNCPTR routine,     /* the routine to call for each hash node */
    int     routineArg   /* arbitrary user-supplied argument */
    )
```

DESCRIPTION This routine calls a user-supplied routine once for each node in the hash table. The routine should be declared as follows:

```
BOOL routine (pNode, arg)
    HASH_NODE * pNode;    /* pointer to a hash table node     */
    int         arg;      /* arbitrary user-supplied argument */
```

The user-supplied routine should return **TRUE** if **hashTblEach( )** is to continue calling it with the remaining nodes, or **FALSE** if it is done and **hashTblEach( )** can exit.

RETURNS **NULL** if traversed whole hash table, or pointer to **HASH_NODE** that hashTblEach ended with.

ERRNO N/A

SEE ALSO **hashLib**

hashTblFind( )

NAME **hashTblFind( )** – find a hash node that matches the specified key

SYNOPSIS

```
HASH_NODE *hashTblFind
    (
    FAST HASH_ID hashId,       /* id of hash table from which to find node */
    HASH_NODE    *pMatchNode,  /* pointer to hash node to match */
    int          keyCmpArg     /* parameter to be passed to key comparator */
    )
```

DESCRIPTION This routine finds the hash node that matches the specified key.

RETURNS pointer to **HASH_NODE**, or **NULL** if no matching hash node is found.

ERRNO N/A

SEE ALSO **hashLib**

hashTblInit( )

NAME **hashTblInit( )** – initialize a hash table

SYNOPSIS

```
STATUS hashTblInit
    (
    HASH_ID hashId,     /* id of hash table to initialize */
    SL_LIST *pTblMem,   /* pointer to memory of sizeLog2 SL_LISTs */
    int      sizeLog2,   /* number of elements in hash table log 2 */
    FUNCPTR keyCmpRtn,   /* function to test keys for equivalence */
    FUNCPTR keyRtn,      /* hashing function to generate hash from key */
    int      keyArg      /* argument to hashing function */
    )
```

DESCRIPTION This routine initializes a hash table. Normally, creation and initialization of the hash table should be done via the routine **hashTblCreate( )**. However, if control over the memory allocation is necessary, this routine is used instead.

All parameters are required with the exception of **keyArg**, which is optional. Refer to **hashTblCreate( )** for a description of parameters.

RETURNS **OK**, or **ERROR** if number of elements is negative, hashId is **NULL**, or the routines passed are **NULL**.

ERRNO N/A

SEE ALSO **hashLib**, **hashTblCreate( )**

hashTblPut( )

NAME **hashTblPut( )** – put a hash node into the specified hash table

SYNOPSIS

```
STATUS hashTblPut
    (
    HASH_ID   hashId,     /* id of hash table in which to put node */
    HASH_NODE *pHashNode  /* pointer to hash node to put in hash table */
    )
```

DESCRIPTION This routine puts the specified hash node in the specified hash table. Identical nodes will be kept in FIFO order in the hash table.

RETURNS **OK**, or **ERROR** if *hashId* is invalid.

ERRNO N/A

SEE ALSO **hashLib**, **hashTblRemove( )**

hashTblRemove( )

NAME

hashTblRemove( ) – remove a hash node from a hash table

SYNOPSIS

```
STATUS hashTblRemove
    (
    HASH_ID   hashId,     /* id of hash table to to remove node from */
    HASH_NODE *pHashNode  /* pointer to hash node to remove */
    )
```

DESCRIPTION

This routine removes the hash node that matches the specified key.

RETURNS

OK, or **ERROR** if *hashId* is invalid.

ERRNO

N/A

SEE ALSO

hashLib

hashTblTerminate( )

NAME

hashTblTerminate( ) – terminate a hash table

SYNOPSIS

```
STATUS hashTblTerminate
    (
    HASH_ID hashId  /* id of hash table to terminate */
    )
```

DESCRIPTION

This routine terminates the specified hash table. The memory for the table is not freed. The hash table is marked as invalid.

RETURNS

OK, or **ERROR** if *hashId* is invalid.

ERRNO

N/A

SEE ALSO

hashLib, **hashTblDestroy( )**, **hashTblDelete( )**

help( )

NAME

help( ) – print a synopsis of selected routines

SYNOPSIS

```
void help (void)
```

DESCRIPTION

This command prints the following list of the calling sequences for commonly used routines, mostly contained in **usrLib**.

```
help                          Print this list
dbgHelp                       Print debug help info
edrHelp                       Print ED&R help info
ioHelp                        Print I/O utilities help info
nfsHelp                       Print nfs help info
netHelp                       Print network help info
rtpHelp                       Print process help info
spyHelp                       Print task histogrammer help info
timexHelp                     Print execution timer help info
h           [n]               Print (or set) shell history
i           [task]            Summary of tasks' TCBs
ti          task              Complete info on TCB for task
sp          adr,args...       Spawn a task, pri=100, opt=0x19, stk=20000
taskSpawn   name,pri,opt,stk,adr,args... Spawn a task
td          task              Delete a task
ts          task              Suspend a task
tr          task              Resume a task
tw          task              Print pending task detailed info
w           [task]            Print pending task info
d           [adr[,nunits[,width]]]   Display memory
m           adr[,width]       Modify memory
mRegs       [reg[,task]]      Modify a task's registers interactively
pc          [task]            Return task's program counter
version                       Print VxWorks version info, and boot line
iam         "user"[,"passwd"]  Set user name and passwd
whoami                        Print user name
devs                          List devices
ld          [syms[,noAbort][,"name"]] Load std in into memory
                               (syms = add symbols to table:
                                -1 = none, 0 = globals, 1 = all)
lkup        ["substr"]        List symbols in system symbol table
lkAddr      address           List symbol table entries near address
checkStack  [task]            List task stack sizes and usage
printErrno  value             Print the name of a status value
period      secs,adr,args...  Spawn task to call function periodically
repeat      n,adr,args...     Spawn task to call function n times (0=forever)
shConfig    ["config"]        Display or set shell configuration variables
strFree     [address]         Free strings allocated within the shell

NOTE: Arguments specifying <task> can be either task ID or name.
```

RETURNS

N/A

ERRNO N/A

SEE ALSO **usrLib**, **ioHelp( )**, **netHelp( )**, **spyHelp( )**, the VxWorks programmer guides.

histLoad( )

NAME **histLoad( )** – load history into the current shell session interpreter(s)

SYNOPSIS

```
void histLoad
    (
    char * loadFile,  /* file path to load the history from */
    BOOL   allInterp  /* whether to save for all interpreters */
    )
```

DESCRIPTION This command loads the shell history for the current shell session. If *allInterp* is set to **TRUE**, the load is done for all registered interpreters, otherwise only the history corresponding to the current interpreter is loaded.

The full path of the history file (including its name) is pointed by *loadFile* which must be **MAX_FILENAME_LENGTH** bytes long at most (including EOS). If *loadFile* is set to **NULL**, the system loads the history from a file named shellHistory.dat in the current directory.

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrShellHistLib**, **shellInterpLib**, the VxWorks programmer guides.

histSave( )

NAME **histSave( )** – save history of the current shell session interpreter(s)

SYNOPSIS

```
void histSave
    (
    char * saveFile,  /* file path to save the history to */
    BOOL   allInterp  /* whether to save for all interpreters */
    )
```

DESCRIPTION This command saves the shell history for the current shell session. If *allInterp* is set to **TRUE**, the save is done for all registered interpreters, otherwise only the current interpreter history is saved.

The full path of the save file (including its name) is pointed to by *saveFile* which must be **MAX_FILENAME_LENGTH** bytes long at most (including EOS). If *saveFile* is set to **NULL**, the history is saved into a file named shellHistory.dat in the current directory.

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrShellHistLib**, **shellInterpLib**, the VxWorks programmer guides.

hookAddToHead( )

NAME **hookAddToHead( )** – add a hook routine at the start of a hook table

SYNOPSIS

```
STATUS hookAddToHead
    (
    void * hook,       /* routine to be added to table */
    void * table[],    /* table to which to add */
    int    maxEntries  /* max entries in table */
    )
```

DESCRIPTION This routine adds a hook routine into a given hook table. The routine is added at the head (i.e. first entry) of the table. Existing hooks are shifted down to make way for the new hook. The last entry of the table is always **NULL**. Hooks are executed from the lowest to highest index of the table. Hence this routine should be used if hooks should be executed in LIFO order (i.e. last hook added executes first). Examples of LIFO hook execution are task delete hooks.

NOTE This routine does not guard against duplicate entries.

RETURNS **OK**, or **ERROR** if hook table is full.

ERRNO **S_hookLib_HOOK_TABLE_FULL**

SEE ALSO **hookLib**

hookAddToTail( )

NAME **hookAddToTail( )** – add a hook routine to the end of a hook table

SYNOPSIS

```
STATUS hookAddToTail
    (
    void * hook,       /* routine to be added to table */
    void * table[],    /* table to which to add */
    int    maxEntries  /* max entries in table */
    )
```

DESCRIPTION This routine adds a hook routine into a given hook table. The routine is added at the first **NULL** entry in the table. In other words new hooks are appended to the list of hooks already present.

NOTE This routine does not guard against duplicate entries.

RETURNS **OK**, or **ERROR** if hook table is full.

ERRNO **S_hookLib_HOOK_TABLE_FULL**

SEE ALSO **hookLib**

hookDelete( )

NAME **hookDelete( )** – delete a hook from a hook table

SYNOPSIS

```
STATUS hookDelete
    (
    void * hook,       /* routine to be deleted from table */
    void * table[],    /* table from which to delete */
    int    maxEntries  /* max entries in table */
    )
```

DESCRIPTION Deletes a previously added hook (if found) from a given hook table. Entries following the deleted hook are moved up to fill the vacant spot created.

RETURNS **OK**, or **ERROR** if hook could not be found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **hookLib**

hookFind( )

NAME **hookFind( )** – Search a hook table for a given hook

SYNOPSIS

```
BOOL hookFind
    (
    void * hook,        /* routine to be deleted from table */
    void * table[],     /* table from which to delete */
    int    maxEntries   /* max entries in table */
    )
```

DESCRIPTION This function searches through a given hook table for a certain hook function. If found **TRUE** is returned, otherwise **FALSE** is returned.

RETURNS **TRUE**, or **FALSE** if the hook was not found.

ERRNO N/A.

SEE ALSO **hookLib**

hookShow( )

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

SYNOPSIS

```
void hookShow
    (
    FUNCPTR table[],     /* table from which to delete */
    int     maxEntries   /* max entries in table */
    )
```

DESCRIPTION Shows the contents of a hook table symbolically.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **hookShow**

hrfsAdvFormat( )

NAME

hrfsAdvFormat( ) – format the HRFS file system using advanced options

SYNOPSIS

```
STATUS hrfsAdvFormat
    (
    char * path,           /* path to format */
    UINT64 diskSize,       /* size of disk in bytes */
    UINT32 blkSize,        /* size of block in bytes */
    UINT32 numInodes,      /* number of Inodes */
    UINT32 majorVersion,   /* file system version to format */
    UINT32 minorVersion,
    UINT32 options         /* misc options */
    )
```

DESCRIPTION

This routine formats a disk or partition referenced by the path to the media.

The *path* argument should be a valid path to the disk or partition to be formatted.

The *diskSize* argument is used to specifiy how many bytes of the media the HRFS file system should occupy. It can be used to prevent HRFS for using the end portion of the media. In general this value should be 0 to specify that the entire media is to be used.

The *blkSize* parameter is used to specify what block size, in byte, HRFS should use. This block size must be a power of 2, greater than the physical sector size, and be within 512 to 8196 bytes inclusively. In general this value should be specified as 0 so the formatter can determine the most efficient block size to use for the media size.

The *numInodes* parameter is used to specify the absolute maximum number of files and directories the file system can ever have. Note this does not include the root directory which the formatter creates automatically. Specifiying a value of 0 will tell the HRFS formatter to allow for the maximum number of files/directories the file system can have based on the amount of data blocks. I.e. One inode per data block.

The *majorVersion* and *minorVersion* parameters are used to specify which particular version of the file system layout should be used when formatting the disk. A value of zero for both the major and minor version is used to indicate that the latest version of the file system should be used.

The *options* parameter is used to specify additional formatting options. It is currently unused

RETURNS

OK on success or **ERROR** on failure.

ERRNO

Not Available

SEE ALSO

hrfsFormatLib

hrfsAdvFormatFd( )

NAME

hrfsAdvFormatFd( ) – format the HRFS file system using advanced options via a file descriptor

SYNOPSIS

```
STATUS hrfsAdvFormatFd
    (
    int    fd,              /* open file descriptor on disk */
    UINT64 diskSize,        /* size of disk in bytes */
    UINT32 blkSize,         /* size of block in bytes */
    UINT32 numInodes,       /* number of Inodes */
    UINT32 majorVersion,    /* file system version to format */
    UINT32 minorVersion,
    UINT32 options          /* misc options */
    )
```

DESCRIPTION

This routine formats a disk or partition referenced by an open file on the media. The file is closed when formatting is complete.

The *fd* argument should be a valid file descriptor representing the root directory of the current file system. This file descriptor will be marked closed and invalid upon return from this function reguardless of outcome.

The *diskSize* argument is used to specifiy how many bytes of the media the HRFS file system should occupy. It can be used to prevent HRFS for using the end portion of the media. In general this value should be 0 to specify that the entire media is to be used.

The *blkSize* parameter is used to specify what block size, in byte, HRFS should use. This block size must be a power of 2, greater than the physical sector size, and be within 512 to 8196 bytes inclusively. In general this value should be specified as 0 so the formatter can determine the most efficient block size to use for the media size.

The *numInodes* parameter is used to specify the absolute maximum number of files and directories the file system can ever have. Note this does not include the root directory which the formatter creates automatically. Specifiying a value of 0 will tell the HRFS formatter to allow for the maximum number of files/directories the file system can have based on the amount of data blocks. I.e. One inode per data block.

The *majorVersion* and *minorVersion* parameters are used to specify which particular version of the file system layout should be used when formatting the disk. A value of zero for both the major and minor version is used to indicate that the latest version of the file system should be used.

The *options* parameter is used to specify additional formatting options. It is currently unused

RETURNS

OK on success or **ERROR** on failure

ERRNO

Not Available

SEE ALSO **hrfsFormatLib**

hrfsAscTime( )

NAME **hrfsAscTime( )** – convert "broken-down" HRFS time to string

SYNOPSIS

```
int hrfsAscTime
    (
    HRFS_TM * pHrfsTm,   /* Buffer contain time to convert in HRFS format */
    char *    pBuffer,   /* Place to write ASCII time format data */
    size_t    bufLength  /* Size of the supplied ASCII time buffer */
    )
```

DESCRIPTION This routine converts the "broken-down" HRFS time pointed to by *pHrfsTm* into a string of the form:

```
SUN SEP 16 01:03:52 1973\en\e0
```

The string is copied into *pBuffer*. Note that the field pHrfsTm->msec is not displayed.

RETURNS the number of bytes copied to *pBuffer*.

ERRNO Not Available

SEE ALSO **hrFsTimeLib**

hrfsChkDsk( )

NAME **hrfsChkDsk( )** – check the HRFS file system

SYNOPSIS

```
STATUS hrfsChkDsk
    (
    char * path,       /* path to check */
    int    verbLevel,  /* verbosity level */
    int    flags       /* additional control information */
    )
```

DESCRIPTION This routine is the HRFS consistency checker. It checks to see if the file system referenced by the path is stable and consistent.

WARNING. This function can only run on an inactive volume. Any currently
opened files will be closed as this routine will eject the current

file system. The volume will also be unaccessible while the consistency checker executes.

The *path* argument should be a valid path to the HRFS formatted disk or partition to be checked.

The *verbLevel* argument is used to specify how much information is outputted to the console. A value of one indicates maximum verbosity. A value of zero indicates minimum verbosity.

The *flags* parameter is used to specify additional control information to the consistency checker. If the **HRFS_CHKDSK_FLAG_UPGRADE** bit is set, the checker will attempt to upgrade the file system to the newest version. All other flags are ignored if this bit is set. If the **HRFS_CHKDSK_FLAG_REWIND_INODE_JOURNAL** bit is set, the checker will attempt to rewind the inode journal. That is, if the inode journal is not empty and is marked as being out of sync with the other disk structures, inodes contained in the inode journal are copied back into the inode table on disk. This has the effect of restoring the inodes to the previous transaction.

RETURNS **OK** if media contains no errors or **ERROR** if one of more problems are detected.

ERRNO Not Available

SEE ALSO **hrfsChkDskLib**

hrfsDevCreate( )

NAME **hrfsDevCreate( )** – create an HRFS device

SYNOPSIS

```
HRFS_DEV_ID hrfsDevCreate
    (
    char *   pDevName,          /* Name of the HRFS device (mount point). */
    device_t xbdId,             /* XBD for the device on which to mount. */
    int      numBufs,           /* # of [struct buf] to allocate. */
    int      maxFiles,          /* Maximum # of simultaneously open files */
    int      defCommitPolicy,   /* Initial commit policy */
    int      defCommitPeriod    /* Initial commit period (if policy is
periodic) */
    )
```

DESCRIPTION This routine creates an HRFS device.

RETURNS **HRFS_DEV_ID** if created and installed in Core I/O, **NULL** if not.

ERRNO
Not Available

SEE ALSO
hrFsLib

hrfsDiskFormat( )

NAME
hrfsDiskFormat( ) – format a disk with HRFS

SYNOPSIS

```
STATUS hrfsDiskFormat
    (
    const char * pDevName,  /* name of the device to initialize */
    int          files,     /* the maximum number of files to support */
    UINT32       majorVer,  /* major version of fs to format */
    UINT32       minorVer,  /* minor version of fs to format */
    UINT32       options    /* formatter options */
    )
```

DESCRIPTION
This command formats a disk and creates the HRFS file system on it. The device must already have been created by the device driver and HRFS format component must be included.

EXAMPLE

```
    -> hrfsDiskFormat "/fd0", 0    /* format "/fd0" with HRFS */
                                   /*allowing maximum files */
    -> hrfsDiskFormat "/fd0", 100  /* format "/fd0" with HRFS */
                                   /*allowing 100 files */
```

RETURNS
OK, or **ERROR** if the device cannot be opened or formatted.

ERRNO
Not Available

SEE ALSO
usrFsLib, **hrFsLib**, the VxWorks programmer guides.

hrfsFormat( )

NAME
hrfsFormat( ) – format the HRFS file system via a path

SYNOPSIS

```
STATUS hrfsFormat
    (
    char * path,      /* path to format */
    UINT64 diskSize,  /* size of disk in bytes */
    UINT32 blkSize,   /* size of block in bytes */
    UINT32 numInodes  /* number of Inodes */
    )
```

DESCRIPTION This routine formats a disk or partition referenced by the path to the media.

The *path* argument should be a valid path to the disk or partition to be formatted.

The *diskSize* argument is used to specifiy how many bytes of the media the HRFS file system should occupy. It can be used to prevent HRFS for using the end portion of the media. In general this value should be 0 to specify that the entire media is to be used.

The *blkSize* parameter is used to specify what block size, in bytes, HRFS should use. This block size must be a power of 2, greater than the physical sector size, and be within 512 to 8196 bytes inclusively. In general this value should be specified as 0 so the formatter can determine the most efficient block size to use for the media size.

The *numInodes* parameter is used to specify the absolute maximum number of files and directories the file system can ever have. Note this does not include the root directory which the formatter creates automatically. Specifiying a value of 0 will tell the HRFS formatter to allow for the maximum number of files/directories the file system can have based on the amount of data blocks. I.e. One inode per data block.

RETURNS **OK** on success or **ERROR** on failure.

ERRNO Not Available

SEE ALSO **hrfsFormatLib**

hrfsFormatFd( )

NAME **hrfsFormatFd( )** – format the HRFS file system via a file descriptor

SYNOPSIS

```
STATUS hrfsFormatFd
    (
    int    fd,         /* open file descriptor on disk */
    UINT64 diskSize,   /* size of disk in bytes */
    UINT32 blkSize,    /* size of block in bytes */
    UINT32 numInodes   /* number of Inodes */
    )
```

DESCRIPTION This routine formats a disk or partition referenced by an open file on the media. The file is closed when formatting is complete.

The *fd* argument should be a valid file descriptor representing the root directory of the current file system. This file descriptor will be marked closed and invalid upon return from this function reguardless of outcome.

The *diskSize* argument is used to specifiy how many bytes of the media the HRFS file system should occupy. It can be used to prevent HRFS for using the end portion of the media. In general this value should be 0 to specify that the entire media is to be used.

The *blkSize* parameter is used to specify what block size, in bytes, HRFS should use. This block size must be a power of 2, greater than the physical sector size, and be within 512 to 8196 bytes inclusively. In general this value should be specified as 0 so the formatter can determine the most efficient block size to use for the media size.

The *numInodes* parameter is used to specify the absolute maximum number of files and directories the file system can ever have. Note this does not include the root directory which the formatter creates automatically. Specifiying a value of 0 will tell the HRFS formatter to allow for the maximum number of files/directories the file system can have based on the amount of data blocks. I.e. One inode per data block.

RETURNS
OK on success or **ERROR** on failure

ERRNO
Not Available

SEE ALSO
hrfsFormatLib

hrfsFormatLibInit( )

NAME
hrfsFormatLibInit( ) – prepare to use the HRFS formatter

SYNOPSIS
```
STATUS hrfsFormatLibInit (void)
```

DESCRIPTION
This routine initializes the HRFS formatter library. This initialization is enabled when the configuration macro **INCLUDE_HRFS_FORMAT** is defined.

RETURNS
OK always

ERRNO
Not Available

SEE ALSO
hrfsFormatLib

hrfsTimeCondense( )

NAME

hrfsTimeCondense( ) – condense time in **HRFS_TM** to time in msec

SYNOPSIS

```
INT64 hrfsTimeCondense
    (
    HRFS_TM * pHrfsTm  /* Pointer to where HRFS_TM format time is stored */
    )
```

DESCRIPTION

This routine condenses the "broken-down" time pointed to by *pHrfsTm* into the number of milliseconds since midnight Jan 1, 1970.

RETURNS

# of milliseconds since midnight Jan 1, 1970

ERRNO

Not Available

SEE ALSO

hrFsTimeLib

hrfsTimeGet( )

NAME

hrfsTimeGet( ) – return # of milliseconds since midnight Jan 1, 1970

SYNOPSIS

```
hrfsTime_t  hrfsTimeGet (void)
```

DESCRIPTION

This routine returns the number of milliseconds since midnight, January 1, 1970.

RETURNS

# of milliseconds since midnight, January 1, 1970

ERRNO

Not Available

SEE ALSO

hrFsTimeLib

hrfsTimeSplit( )

NAME

hrfsTimeSplit( ) – split time in msec into **HRFS_TM** format

SYNOPSIS

```
STATUS hrfsTimeSplit
    (
    INT64     milliSeconds,  /* milliseconds to convert to HRFS_TM format */
    HRFS_TM * pHrfsTm        /* Buffer to store HRFS_TM format time */
    )
```

DESCRIPTION This routine splits the time specified in milliseconds into the "broken-down" format of the **HRFS_TM** structure. Should the equivalent number of seconds exceed what can be represent by a signed integer, this routine will currently return **ERROR**, and the split will not have occurred.

RETURNS **OK** success, or **ERROR** if the split did not occur

ERRNO Not Available

SEE ALSO **hrFsTimeLib**

hrfsUpgrade( )

NAME **hrfsUpgrade( )** – upgrade the HRFS file system to the latest version

SYNOPSIS

```
STATUS hrfsUpgrade
    (
    char *path  /* path to upgrade */
    )
```

DESCRIPTION This routine is the HRFS consistency checker. It checks to see if the file system referenced by the path is stable and consistent.

WARNING. This function can only run on an inactive volume. Any currently
opened files will closed as this routine will eject the current
file system. The volume will also be unaccessible while the
consistency checker executes.

The *path* argument should be a valid path to the HRFS formatted disk or partition to be checked.

RETURNS **OK** if media was upgraded without errors. **ERROR** if one of more
problems are detected.

ERRNO Not Available

SEE ALSO **hrfsChkDskLib**

i( )

NAME

i( ) – print a summary of each task's TCB

SYNOPSIS

```
void i
    (
    int taskNameOrId  /* task name or task ID, 0 = summarize all */
    )
```

DESCRIPTION

This command displays a synopsis of all the tasks in the system. The **ti( )** routine provides more complete information on a specific task.

Both **i( )** and **ti( )** use **taskShow( )**; see the documentation for **taskShow( )** for a description of the output format.

EXAMPLE

```
-> i

  NAME       ENTRY      TID      PRI  STATUS    PC       SP      ERRNO  DELAY
---------- ---------- -------- --- -------- -------- -------- ------- -----
tExcTask   excTask    602adf00   0 PEND     60164ad0 602add08       0     0
tLogTask   logTask    602b56c8   0 PEND     60164ad0 602b54d0       0     0
tShell0    shellTask  60351ba8   1 READY    6015fe68 6034fde8       0     0
tWdbTask   wdbTask    60338308   3 PEND     601579f4 60337ff0       0     0
tNetTask   netTask    602bf6a8  50 PEND     601579f4 602bf4e0       0     0
value = 0 = 0x0
```

CAVEAT

This command should be used only as a debugging aid, since the information is obsolete by the time it is displayed.

SMP CONSIDERATIONS

This command displays a "CPU #" column instead of the "DELAY" column. The "CPU #" column provides information on the CPU a task is executing on, or "-" if a task is not running on a CPU.

SMP EXAMPLE

```
-> i

 NAME         ENTRY        TID     PRI   STATUS      PC        SP      ERRNO  CPU
#
----------  ------------ -------- --- ---------- -------- -------- -------
-----
tExcTask    186438         247210   0 PEND         1e0198   249390       0
-
tJobTask    187390         2a5830   0 PEND         1e0198   2a5770       0
-
tLogTask    logTask        2a8990   0 PEND         1dd86c   2a8870       0
-
tNbioLog    188848         2ac220   0 PEND         1e0198   2ac110       0
-
tShell0     shellTask      2be530   1 READY        1e8ec8   2bc780       0
1
```

```
miiBusMoni> 134794         29d010 254 DELAY        1e65a0   29cf80       0
-
tIdleTask0  idleTaskEntr   24d010 287 READY        1dfb1c   24cf90       0
0
tIdleTask1  idleTaskEntr   250630 287 READY        1dfb28   2505b0       0
-
value = 0 = 0x0
```

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **ti( )**, **taskShow( )**, the VxWorks programmer guides.

i0( )

NAME **i0( )** – return the contents of register i0 (also i1-i7) (SimSolaris)

SYNOPSIS

```
int i0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of in register i0 from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all in registers (i0 - i7): **i0( )** - **i7( )**.

The frame pointer is accessed via i6.

RETURNS The contents of register i0 (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

i8042vxbRegister( )

NAME **i8042vxbRegister( )** – register i8042vxb driver

SYNOPSIS

```
void i8042vxbRegister(void)
```

DESCRIPTION This routine registers the i8042vxb driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO **vxbI8042Kbd**

ichAtaBlkRW( )

NAME **ichAtaBlkRW( )** – read or write sectors to a ATA/IDE disk.

SYNOPSIS

```
STATUS ichAtaBlkRW
    (
    ATA_DEV  *pDev,
    sector_t startBlk,
    UINT32   nBlks,
    char     *pBuf,
    int      direction
    )
```

DESCRIPTION Read or write sectors to a ATA/IDE disk. *startBlk* is the start Block, *nBlks* is the number of blocks, *pBuf* is data buffer pointer and *direction* is the direction either to read or write. It should be **O_WRONLY** for data write to drive or **O_RDONLY** for read data from drive.

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaCmd( )

NAME **ichAtaCmd( )** – issue a RegisterFile command to ATA/ATAPI device.

SYNOPSIS

```
STATUS ichAtaCmd
```

```
    (
    int ctrl,   /* Controller number. 0 or 1 */
    int drive,  /* Drive number.      0 or 1 */
    int cmd,    /* Command Register          */
    int arg0,   /* argument0 */
    int arg1,   /* argument1 */
    int arg2,   /* argument2 */
    int arg3,   /* argument3 */
    int arg4,   /* argument4 */
    int arg5    /* argument5 */
    )
```

DESCRIPTION

This function executes ATA command to ATA/ATAPI devices specified by arguments *ctrl* and *drive*. *cmd* is command to be executed and other arguments *arg0* to *arg5* are interpreted for differently in each case depending on the *cmd* command. Some commands (like **ATA_CMD_SET_FEATURE**) have sub commands the case in which *arg0* is interpreted as subcommand and *arg1* is subcommand specific.

In general these arguments *arg0* to *arg5* are interpreted as command registers of the device as mentioned below.

arg0	- Feature Register
arg1	- Sector count
arg2	- Sector number
arg3	- CylLo
arg4	- CylHi
arg5	- sdh Register

As these registers are interpreted for different purpose for each command, arguments are not named after registers.

The following commands are valid in this function and the validity of each argument for different commands. Each command is tabulated in the form

```
---------------------------------------------------------------------
COMMAND
      ARG0     |   ARG1     |   ARG2    |   ARG3    |   ARG4    |   ARG5
---------------------------------------------------------------------

ATA_CMD_INITP
       0             0            0           0           0          0

ATA_CMD_RECALIB
       0             0            0           0           0          0

ATA_PI_CMD_SRST
       0             0            0           0           0          0

ATA_CMD_EXECUTE_DEVICE_DIAGNOSTIC
       0             0            0           0           0          0

ATA_CMD_SEEK
      cylinder    head            0           0           0          0
or   LBA high     LBA low
```

```
ATA_CMD_SET_FEATURE
      FR          SC            0          0          0          0
  (SUBCOMMAND)  (SubCommand
              Specific Value)

ATA_CMD_SET_MULTI
  sectors per block  0          0          0          0          0

ATA_CMD_IDLE
      SC             0          0          0          0          0
   (Timer Period)

ATA_CMD_STANDBY
      SC             0          0          0          0          0
   (Timer Period)

ATA_CMD_STANDBY_IMMEDIATE
       0             0            0          0          0          0

ATA_CMD_SLEEP
       0             0            0          0          0          0

ATA_CMD_CHECK_POWER_MODE
       0             0            0          0          0          0

ATA_CMD_IDLE_IMMEDIATE
       0             0            0          0          0          0

ATA_CMD_SECURITY_DISABLE_PASSWORD
   ATA_ZERO     ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_SECURITY_ERASE_PREPARE
       0             0            0          0          0          0

ATA_CMD_SECURITY_ERASE_UNIT
   ATA_ZERO     ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_SECURITY_FREEZE_LOCK
       0             0            0          0          0          0

ATA_CMD_SECURITY_SET_PASSWORD
       0             0            0          0          0          0

ATA_CMD_SECURITY_UNLOCK
       0             0            0          0          0          0

ATA_CMD_SMART  (not implemented)
      FR          SC          SN        ATA_ZERO   ATA_ZERO     ATA_ZERO
 (SUBCOMMAND) (SubCommand   (SubCommand
            Specific Value)  Specific Value)

ATA_CMD_GET_MEDIA_STATUS
       0             0            0          0          0          0

ATA_CMD_MEDIA_EJECT
       0             0            0          0          0          0

ATA_CMD_MEDIA_LOCK
       0             0            0          0          0          0

ATA_CMD_MEDIA_UNLOCK
       0             0            0          0          0          0

ATA_CMD_CFA_ERASE_SECTORS
       0             0            0          0          0          0

ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE
    ATA_ZERO        SC      ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE
    ATA_ZERO        SC      ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO
```

```
ATA_CMD_CFA_TRANSLATE_SECTOR
    ATA_ZERO    ATA_ZERO        SN         cylLo     cylHi       DH

ATA_CMD_CFA_REQUEST_EXTENDED_ERROR_CODE
    ATA_ZERO    ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_SET_MAX
      FR        ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO
    (SUBCOMMAND)
```

The following are the subcommands valid for **ATA_CMD_SET_MAX** and are tabulated as below

```
---------------------------------------------------------------------
SUBCOMMAND(in ARG0)
      ARG1    |       ARG2    |       ARG3    |       ARG4    |      ARG5
---------------------------------------------------------------------

ATA_SUB_SET_MAX_ADDRESS
      SC      sector no   cylLo       cylHi       head + modebit
(SET_MAX_VOLATILE
      or
SET_MAX_NON_VOLATILE)

ATA_SUB_SET_MAX_SET_PASS
      ATA_ZERO        ATA_ZERO        ATA_ZERO        ATA_ZERO       ATA_ZERO

ATA_SUB_SET_MAX_LOCK
      ATA_ZERO        ATA_ZERO        ATA_ZERO        ATA_ZERO       ATA_ZERO

ATA_SUB_SET_MAX_UNLOCK
      ATA_ZERO        ATA_ZERO        ATA_ZERO        ATA_ZERO       ATA_ZERO

ATA_SUB_SET_MAX_FREEZE_LOCK
      ATA_ZERO        ATA_ZERO        ATA_ZERO        ATA_ZERO       ATA_ZERO
```

In **ATA_CMD_SET_FEATURE** subcommand only arg0 and arg1 are valid, all other are **ATA_ZERO**.

```
------------------------------------------------------
SUBCOMMAND(ARG0)                          ARG1
------------------------------------------------------

ATA_SUB_ENABLE_8BIT                       ATA_ZERO

ATA_SUB_ENABLE_WCACHE                     ATA_ZERO

ATA_SUB_SET_RWMODE                        mode
                                   (see page no 168 table 28 in atapi Spec5 )
ATA_SUB_ENB_ADV_POW_MNGMNT                0x90

ATA_SUB_ENB_POW_UP_STDBY                  ATA_ZERO

ATA_SUB_POW_UP_STDBY_SPIN                 ATA_ZERO

ATA_SUB_BOOTMETHOD                        ATA_ZERO

ATA_SUB_ENA_CFA_POW_MOD1                  ATA_ZERO

ATA_SUB_DISABLE_NOTIFY                    ATA_ZERO

ATA_SUB_DISABLE_RETRY                     ATA_ZERO

ATA_SUB_SET_LENGTH                        ATA_ZERO

ATA_SUB_SET_CACHE                         ATA_ZERO

ATA_SUB_DISABLE_LOOK                      ATA_ZERO
```

ATA_SUB_ENA_INTR_RELEASE	ATA_ZERO
ATA_SUB_ENA_SERV_INTR	ATA_ZERO
ATA_SUB_DISABLE_REVE	ATA_ZERO
ATA_SUB_DISABLE_ECC	ATA_ZERO
ATA_SUB_DISABLE_8BIT	ATA_ZERO
ATA_SUB_DISABLE_WCACHE	ATA_ZERO
ATA_SUB_DIS_ADV_POW_MNGMT	ATA_ZERO
ATA_SUB_DISB_POW_UP_STDBY	ATA_ZERO
ATA_SUB_ENABLE_ECC	ATA_ZERO
ATA_SUB_BOOTMETHOD_REPORT	ATA_ZERO
ATA_SUB_DIS_CFA_POW_MOD1	ATA_ZERO
ATA_SUB_ENABLE_NOTIFY	ATA_ZERO
ATA_SUB_ENABLE_RETRY	ATA_ZERO
ATA_SUB_ENABLE_LOOK	ATA_ZERO
ATA_SUB_SET_PREFETCH	ATA_ZERO
ATA_SUB_SET_4BYTES	ATA_ZERO
ATA_SUB_ENABLE_REVE	ATA_ZERO
ATA_SUB_DIS_INTR_RELEASE	ATA_ZERO
ATA_SUB_DIS_SERV_INTR	ATA_ZERO

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaConfig( )

NAME **ichAtaConfig( )** – configure an ATA drive (hard disk or cdrom drive)

SYNOPSIS

```
STATUS ichAtaConfig
    (
    int  ctrl,      /* 0: primary address, 1: secondary address */
    int  drive,     /* drive number of hard disk (0 or 1) */
    char *devNames  /* mount points for each partition */
    )
```

DESCRIPTION This routine configures an ATA hard disk. Parameters:

drive
the drive number of the hard disk; 0 is **C:** and 1 is **D:**.

devName
the mount point for all partitions which are expected to be present on the disk, separated with commas, for example "/ata0,/ata1" or "C:,D:". Blanks are not allowed in this string.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**, **src/config/usrAta.c**, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486/Pentium*

ichAtaConfigInit( )

NAME **ichAtaConfigInit( )** – intialize the hard disk driver

SYNOPSIS

```
void ichAtaConfigInit (void)
```

DESCRIPTION This routine is called from **usrConfig.c** to initialize the hard drive.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaCtrlReset( )

NAME **ichAtaCtrlReset( )** – reset the specified ATA/IDE disk controller

SYNOPSIS

```
STATUS ichAtaCtrlReset
    (
    int ctrl
    )
```

DESCRIPTION This routine resets the ATA controller specified by ctrl. The device control register is written with SRST=1

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaDevCreate( )

NAME **ichAtaDevCreate( )** – create a device for a ATA/IDE disk

SYNOPSIS

```
BLK_DEV * ichAtaDevCreate
    (
    int    ctrl,      /* ATA controller number, 0 is the primary controller
*/
    int    drive,     /* ATA drive number, 0 is the master drive */
    UINT32 nBlocks,   /* number of blocks on device, 0 = use entire disc */
    UINT32 blkOffset  /* offset BLK_DEV nBlocks from the start of the drive
*/
    )
```

DESCRIPTION This routine creates a device for a specified ATA/IDE or ATAPI CDROM disk.

ctrl is a controller number for the ATA controller; the primary controller is 0. The maximum is specified via **ATA_MAX_CTRLS**.

drive is the drive number for the ATA hard drive; the master drive is 0. The maximum is specified via **ATA_MAX_DRIVES**.

The *nBlocks* parameter specifies the size of the device in blocks. If *nBlocks* is zero, the whole disk is used.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the hard disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.)

RETURNS A pointer to a block device structure (**BLK_DEV**) or **NULL** if memory cannot be allocated for the device structure.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**, **dosFsMkfs( )**, **dosFsDevInit( )**, **rawFsDevInit( )**

ichAtaDevIdentify( )

NAME

ichAtaDevIdentify( ) – identify device

SYNOPSIS

```
STATUS ichAtaDevIdentify
    (
    int ctrl,
    int dev
    )
```

DESCRIPTION

This routine checks whether the device is connected to the controller, if it is, this routine determines drive type. The routine set **type** field in the corresponding **ATA_DRIVE** structure. If device identification failed, the routine set **state** field in the corresponding **ATA_DRIVE** structure to **ATA_DEV_NONE**.

RETURNS

TRUE if a device present, **FALSE** otherwise

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorage

ichAtaDmaRW( )

NAME

ichAtaDmaRW( ) – read/write a number of sectors on the current track in DMA mode

SYNOPSIS

```
STATUS ichAtaDmaRW
    (
    int      ctrl,
    int      drive,
    UINT32   cylinder,
    UINT32   head,
    UINT32   sector,
    void *   buffer,
    UINT32   nSecs,
    int      direction,
    sector_t startBlk
    )
```

DESCRIPTION

Read/write a number of sectors on the current track in DMA mode

RETURNS

OK, **ERROR** if the command didn't succeed.

ERRNO

Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaDmaToggle( )

NAME **ichAtaDmaToggle( )** – turn on or off an individual controllers dma support

SYNOPSIS

```
void ichAtaDmaToggle
    (
    int ctrl
    )
```

DESCRIPTION This routine lets you toggle the DMA setting for an individual controller. The controller number is passed in as a parameter, and the current value is toggled.

RETURNS **OK**, or **ERROR** if the parameters are invalid.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtaDrv( )

NAME **ichAtaDrv( )** – Initialize the ATA driver

SYNOPSIS

```
STATUS ichAtaDrv
    (
    int ctrl,        /* controller no. 0,1   */
    int drives,      /* number of drives 1,2 */
    int vector,      /* interrupt vector     */
    int level,       /* interrupt level      */
    int configType,  /* configuration type   */
    int semTimeout,  /* timeout seconds for sync semaphore */
    int wdgTimeout   /* timeout seconds for watch dog      */
    )
```

DESCRIPTION This routine initializes the ATA/ATAPI device driver, initializes IDE host controller and sets up interrupt vectors for requested controller. This function must be called once for each controller, before any access to drive on the controller, usually which is called by **usrRoot( )** in **usrConfig.c**.

If it is called more than once for the same controller, it returns **OK** with a message display **Host controller already initialized** , and does nothing as already required initialization is done.

Additionally it identifies devices available on the controller and initializes depending on the type of the device (ATA or ATAPI). Initialization of device includes reading parameters of the device and configuring to the defaults.

RETURNS **OK**, or **ERROR** if initialization fails.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**, **ichAtaDevCreate( )**

ichAtaDumptest( )

NAME **ichAtaDumptest( )** – a quick test of the dump functionality for ATA driver

SYNOPSIS

```
void ichAtaDumptest
    (
    device_t d,
    sector_t sector,
    UINT32   blocks,
    char     *data
    )
```

DESCRIPTION *device_t* device id of the device to dump to. This can be any XBD
device. Could be the XBD of the disk device itself, or
could be the xbd of a partition overlayed on the drive.

sector sector offset to begin dump relative to start of xbd. *blocks* number of blocks to dump to device **data* buffer that contains data to dump

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtaInit( )

NAME **ichAtaInit( )** – initialize ATA device.

SYNOPSIS
```
STATUS ichAtaInit
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This routine issues a soft reset command to ATA device for initialization.

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaParamRead( )

NAME **ichAtaParamRead( )** – Read drive parameters

SYNOPSIS
```
STATUS ichAtaParamRead
    (
    int  ctrl,
    int  drive,
    void *buffer,
    int  command
    )
```

DESCRIPTION Read drive parameters.

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaPiInit( )

NAME

ichAtaPiInit( ) – init a ATAPI CD-ROM disk controller

SYNOPSIS

```
STATUS ichAtaPiInit
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This routine resets a ATAPI CD-ROM disk controller.

RETURNS

OK, **ERROR** if the command didn't succeed.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorage

ichAtaRW( )

NAME

ichAtaRW( ) – read/write a data from/to required sector.

SYNOPSIS

```
STATUS ichAtaRW
    (
    int       ctrl,
    int       drive,
    UINT32    cylinder,
    UINT32    head,
    UINT32    sector,
    void    * buffer,
    UINT32    nSecs,
    int       direction,
    sector_t  startBlk
    )
```

DESCRIPTION

Read/write a number of sectors on the current track

RETURNS

OK, **ERROR** if the command didn't succeed.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorage

ichAtaRawio( )

NAME

ichAtaRawio( ) – do raw I/O access

SYNOPSIS

```
STATUS ichAtaRawio
    (
    int     ctrl,
    int     drive,
    ATA_RAW *pAtaRaw
    )
```

DESCRIPTION

This routine is called to perform raw I/O access.

drive is a drive number for the hard drive: it must be 0 or 1.

The *pAtaRaw* is a pointer to the structure **ATA_RAW** which is defined in **ichAtaDrv.h**.

RETURNS

OK, or **ERROR** if the parameters are not valid.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorage

ichAtaShow( )

NAME

ichAtaShow( ) – show the ATA/IDE disk parameters

SYNOPSIS

```
STATUS ichAtaShow
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This routine shows the ATA/IDE disk parameters. Its first argument is a controller number, 0 or 1; the second argument is a drive number, 0 or 1.

RETURNS

OK, or **ERROR** if the parameters are invalid.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorageShow

ichAtaShowInit( )

NAME **ichAtaShowInit( )** – initialize the ATA/IDE disk driver show routine

SYNOPSIS

```
STATUS ichAtaShowInit (void)
```

DESCRIPTION This routine links the ATA/IDE disk driver show routine 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_ATA_SHOW**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtaStatusChk( )

NAME **ichAtaStatusChk( )** – Check status of drive and compare to requested status.

SYNOPSIS

```
STATUS ichAtaStatusChk
    (
    ATA_CTRL  * pCtrl,
    UINT8       mask,
    UINT8       status
    )
```

DESCRIPTION Wait until the drive is ready.

RETURNS **OK**, **ERROR** if the drive status check times out.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtaXbdDevCreate( )

NAME

ichAtaXbdDevCreate( ) – create an XBD device for a ATA/IDE disk

SYNOPSIS

```
device_t ichAtaXbdDevCreate
    (
    int          ctrl,       /* ATA controller number, 0 is the primary
controller */
    int          drive,      /* ATA drive number, 0 is the master drive */
    UINT32       nBlocks,    /* number of blocks on device, 0 = use entire
disc */
    UINT32       blkOffset,  /* offset BLK_DEV nBlocks from the start of the
drive */
    const char * name        /* name of xbd device to create */
    )
```

DESCRIPTION

Use the existing code to create a standard block dev device, then create an XBD device associated with the BLKDEV.

RETURNS

a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO

vxbIntelIchStorage

ichAtaXbdRawio( )

NAME

ichAtaXbdRawio( ) – do raw I/O access

SYNOPSIS

```
STATUS ichAtaXbdRawio
    (
    device_t device,
    sector_t sector,
    UINT32   numSecs,
    char     *data,
    int      direction
    )
```

DESCRIPTION

This routine is called to perform raw I/O access.

device is the XBD device identifier for the drive *sector* starting sector for I/O operation *numSecs* number of sectors to read/write *data* pointer to data buffer *dir* read or write

The *pAtaRaw* is a pointer to the structure **ATA_RAW** which is defined in **ichAtaDrv.h**.

RETURNS **OK**, or **ERROR** if the parameters are not valid.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiBytesPerSectorGet( )

NAME **ichAtapiBytesPerSectorGet( )** – get the number of Bytes per sector.

SYNOPSIS

```
UINT16 ichAtapiBytesPerSectorGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the number of Bytes per sector. This function will return correct values for drives of ATA/ATAPI-4 or less as this field is retired for the drives compliant to ATA/ATAPI-5 or higher.

RETURNS Bytes per sector.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiBytesPerTrackGet( )

NAME **ichAtapiBytesPerTrackGet( )** – get the number of Bytes per track.

SYNOPSIS

```
UINT16 ichAtapiBytesPerTrackGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the number of Bytes per track. This function will return correct values for drives of ATA/ATAPI-4 or less as this feild is retired for the drives compliant to ATA/ATAPI-5 or higher.

RETURNS Bytes per track.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCtrlMediumRemoval( )

NAME **ichAtapiCtrlMediumRemoval( )** – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command

SYNOPSIS

```
STATUS ichAtapiCtrlMediumRemoval
    (
    ATA_DEV   * pAtapiDev,
    int         arg0
    )
```

DESCRIPTION This function issues a command to drive to PREVENT or ALLOW MEDIA removal. Argument *arg0* selects to **LOCK_EJECT** or **UNLOCK_EJECT**.

To lock media eject *arg0* should be **LOCK_EJECT** To unload media eject *arg0* should be **UNLOCK_EJECT**

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiCurrentCylinderCountGet( )

NAME **ichAtapiCurrentCylinderCountGet( )** – get logical number of cylinders in the drive.

SYNOPSIS

```
UINT16 ichAtapiCurrentCylinderCountGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the number of logical cylinders in the drive. This value represents the no of cylinders that can be addressed.

RETURNS Cylinder count.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentHeadCountGet( )

NAME **ichAtapiCurrentHeadCountGet( )** – get the number of read/write heads in the drive.

SYNOPSIS

```
UINT8 ichAtapiCurrentHeadCountGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the number of heads in the drive from device structure.

RETURNS Number of heads.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentMDmaModeGet( )

NAME **ichAtapiCurrentMDmaModeGet( )** – get the enabled Multi word DMA mode.

SYNOPSIS

```
UINT8 ichAtapiCurrentMDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive MDMA mode enable in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. The following bit is set for corresponding mode selected.

- Bit2 Multi DMA mode 2 is Selected
- Bit1 Multi DMA mode 1 is Selected
- Bit0 Multi DMA mode 0 is Selected

RETURNS Enabled Multi word DMA mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentPioModeGet( )

NAME **ichAtapiCurrentPioModeGet( )** – get the enabled PIO mode.

SYNOPSIS

```
UINT8 ichAtapiCurrentPioModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive current PIO mode enabled in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure.

RETURNS Enabled PIO mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentRwModeGet( )

NAME **ichAtapiCurrentRwModeGet( )** – get the current Data transfer mode.

SYNOPSIS

```
UINT8 ichAtapiCurrentRwModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the current Data transfer mode if it is PIO 0,1,2,3,4 mode, SDMA 0,1,2 mode, MDMA 0,1,2 mode or UDMA 0,1,2,3,4,5 mode.

RETURNS current PIO mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentSDmaModeGet( )

NAME **ichAtapiCurrentSDmaModeGet( )** – get the enabled Single word DMA mode.

SYNOPSIS

```
UINT8 ichAtapiCurrentSDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive SDMA mode enable in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure

RETURNS Enabled Single word DMA mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCurrentUDmaModeGet( )

NAME **ichAtapiCurrentUDmaModeGet( )** – get the enabled Ultra DMA mode.

SYNOPSIS

```
UINT8 ichAtapiCurrentUDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive UDMA mode enable in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure The following bit is set for corresponding mode selected.

- Bit4 Ultra DMA mode 4 is Selected
- Bit3 Ultra DMA mode 3 is Selected
- Bit2 Ultra DMA mode 2 is Selected
- Bit1 Ultra DMA mode 1 is Selected
- Bit0 Ultra DMA mode 0 is Selected

RETURNS Enabled Ultra DMA mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiCylinderCountGet( )

NAME **ichAtapiCylinderCountGet( )** – get the number of cylinders in the drive.

SYNOPSIS

```
UINT16 ichAtapiCylinderCountGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get cyclinder count of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure.

RETURNS Cylinder count.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiDriveSerialNumberGet( )

NAME

ichAtapiDriveSerialNumberGet( ) – get the drive serial number.

SYNOPSIS

```
char * ichAtapiDriveSerialNumberGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function is used to get drive serial number of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 20 bytes length which contains serial number in ascii.

RETURNS

Drive serial number.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorageShow

ichAtapiDriveTypeGet( )

NAME

ichAtapiDriveTypeGet( ) – get the drive type.

SYNOPSIS

```
UINT8 ichAtapiDriveTypeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function routine will return the type of the drive if it is CD-ROM or Printer etc. The following table indicates the type depending on the return value.

0x00h	Direct-access device
0x01h	Sequential-access device
0x02h	Printer Device
0x03h	Processor device
0x04h	Write-once device
0x05h	CD-ROM device
0x06h	Scanner device
0x07h	Optical memory device
0x08h	Medium Change Device
0x09h	Communications device

0x0Ch	Array Controller Device
0x0Dh	Encloser Services Device
0x0Eh	Reduced Block Command Devices
0x0Fh	Optical Card Reader/Writer Device
0x1Fh	Unknown or no device type

RETURNS drive type.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiFeatureEnabledGet( )

NAME **ichAtapiFeatureEnabledGet( )** – get the enabled features.

SYNOPSIS

```
UINT32 ichAtapiFeatureEnabledGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive Features Enabled by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a 32 bit value whose bits represents the features Enabled. The following table gives the cross reference for the bits.

Bit 21	Power-up in Standby Feature
Bit 20	Removable Media Status Notification Feature
Bit 19	Adavanced Power Management Feature
Bit 18	CFA Feature
Bit 10	Host protected Area Feature
Bit 4	Packet Command Feature
Bit 3	Power Management Feature
Bit 2	Removable Media Feature
Bit 1	Security Mode Feature
Bit 0	SMART Feature

RETURNS enabled features.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiFeatureSupportedGet( )

NAME **ichAtapiFeatureSupportedGet( )** – get the features supported by the drive.

SYNOPSIS

```
UINT32 ichAtapiFeatureSupportedGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive Feature supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a 32 bit value whose bits represents the features supported. The following table gives the cross reference for the bits.

Bit 21	Power-up in Standby Feature
Bit 20	Removable Media Status Notification Feature
Bit 19	Adavanced Power Management Feature
Bit 18	CFA Feature
Bit 10	Host protected Area Feature
Bit 4	Packet Command Feature
Bit 3	Power Management Feature
Bit 2	Removable Media Feature
Bit 1	Security Mode Feature
Bit 0	SMART Feature

RETURNS Supported features.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiFirmwareRevisionGet( )

NAME **ichAtapiFirmwareRevisionGet( )** – get the firm ware revision of the drive.

SYNOPSIS

```
char * ichAtapiFirmwareRevisionGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive Firmware revision of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 8 bytes length which contains serial number in ascii.

RETURNS firmware revision.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiHeadCountGet( )

NAME **ichAtapiHeadCountGet( )** – get the number heads in the drive.

SYNOPSIS

```
UINT8 ichAtapiHeadCountGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get head count of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure.

RETURNS Number of heads in the drive.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiInit( )

NAME **ichAtapiInit( )** – init ATAPI CD-ROM disk controller

SYNOPSIS

```
STATUS ichAtapiInit
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This routine resets the ATAPI CD-ROM disk controller.

RETURNS **OK**, **ERROR** if the command didn't succeed.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiIoctl( )

NAME **ichAtapiIoctl( )** – Control the drive.

SYNOPSIS

```
STATUS ichAtapiIoctl
    (
    int                  function,  /* The IO operation to do */
    int                  ctrl,      /* Controller number of the drive */
    int                  drive,     /* Drive number */
    int         password [16],      /* Password to set. NULL if not
applicable */
    int                  arg0,      /* 1st arg to pass. NULL if not
applicable */
    UINT32   *           arg1,      /* Ptr to 2nd arg.  NULL if not
applicable */
    UINT8    **          ppBuf      /* The data buffer */
    )
```

DESCRIPTION This routine is used to control the drive like setting the password, putting in power save mode, locking/unlocking the drive, ejecting the medium etc. The argument *function* defines the ioctl command, *password*, and integer array is the password required or set password value for some commands. Arguments *arg0*, pointer *arg1*, pointer to pointer buffer *ppBuf* are commad specific.

The following commands are supported for various functionality.

IOCTL_DIS_MASTER_PWD
Disable the master password. where 4th parameter is the master password.

IOCTL_DIS_USER_PWD
Disable the user password.

IOCTL_ERASE_PREPARE
Prepare the drive for erase incase the user password lost, and it is in max security mode.

IOCTL_ENH_ERASE_UNIT_USR
Erase in enhanced mode supplying the user password.

IOCTL_ENH_ERASE_UNIT_MSTR
Erase in enhanced mode supplying the master password.

IOCTL_NORMAL_ERASE_UNIT_MSTR
Erase the drive in normal mode supplying the master password.

IOCTL_NORMAL_ERASE_UNIT_USR
Erase the drive in normal mode supplying the user password.

IOCTL_FREEZE_LOCK
Freeze lock the drive.

IOCTL_SET_PASS_MSTR
Set the master password.

IOCTL_SET_PASS_USR_MAX
Set the user password in Maximum security mode.

IOCTL_SET_PASS_USR_HIGH
Set the user password in High security mode.

IOCTL_UNLOCK_MSTR
Unlock the master password.

IOCTL_UNLOCK_USR
Unlock the user password.

IOCTL_CHECK_POWER_MODE
Find the drive power saving mode.

IOCTL_IDLE_IMMEDIATE
Idle the drive immediatly. this will get the drive from the standby or active mode to idle mode immediatly.

IOCTL_SLEEP
Set the drive in sleep mode. this is the highest power saving mode. to return to the normal active or IDLE mode, drive need an hardware reset or power on reset or device reset command.

IOCTL_STANDBY_IMMEDIATE
Standby the drive immediatly.

IOCTL_EJECT_DISK
Eject the media of an ATA drive. Use IOsystem ioctl function for ATAPI drive.

IOCTL_GET_MEDIA_STATUS
Find the media status.

IOCTL_ENA_REMOVE_NOTIFY
Enable the drive's removable media notification feature set.

The following table describes these arguments validity. These are tabulated in the following form

```
----------------------------------------------------------------------
FUNCTION
password [16]         arg0            *arg1               **ppBuf
----------------------------------------------------------------------

IOCTL_DIS_MASTER_PWD
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_DIS_USER_PWD
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_ERASE_PREPARE
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_ENH_ERASE_UNIT_USR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_ENH_ERASE_UNIT_MSTR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_NORMAL_ERASE_UNIT_MSTR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_NORMAL_ERASE_UNIT_USR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_FREEZE_LOCK
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SET_PASS_MSTR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SET_PASS_USR_MAX
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SET_PASS_USR_HIGH
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_UNLOCK_MSTR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_UNLOCK_USR
password            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_READ_NATIVE_MAX_ADDRESS    - it returns address in <arg1>
ATA_ZERO         (ATA_SDH_IBM or    LBA/CHS add           ATA_ZERO
                  ATA_SDH_LBA )   ( LBA 27:24 / Head
                                    LBA 23:16 / cylHi
                                    LBA 15:8  / cylLow
                                    LBA 7:0   / sector no )

IOCTL_SET_MAX_ADDRESS           - <arg1> is pointer to LBA address
ATA_ZERO    SET_MAX_VOLATILE or     LBA address           ATA_ZERO
            SET_MAX_NON_VOLATILE

IOCTL_SET_MAX_SET_PASS
password            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_SET_MAX_LOCK
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_SET_MAX_UNLOCK
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_SET_MAX_FREEZE_LOCK
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO
```

```
IOCTL_CHECK_POWER_MODE           - returns power mode in <arg1>
ATA_ZERO            ATA_ZERO          returns power        ATA_ZERO
                                      mode
         power modes  :-1)    0x00  Device in standby mode
                        2)    0x80  Device in Idle mode
                3)   0xff  Device in Active or Idle mode

IOCTL_IDLE_IMMEDIATE
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_SLEEP
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_STANDBY_IMMEDIATE
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_ENB_POW_UP_STDBY
ATA_ZERO            ATA_ZERO         ATA_ZERO             ATA_ZERO

IOCTL_ENB_SET_ADV_POW_MNGMNT
ATA_ZERO                arg0         ATA_ZERO             ATA_ZERO

 NOTE:- arg0 value - 1). for minimum power consumption with standby 0x01h
                     2). for minimum power consumption without standby 0x01h
                     3). for maximum performance  0xFEh

IOCTL_DISABLE_ADV_POW_MNGMNT
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_EJECT_DISK
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_LOAD_DISK
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_MEDIA_LOCK
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_MEDIA_UNLOCK
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_GET_MEDIA_STATUS      - returns status in <arg1>
ATA_ZERO            ATA_ZERO             status           ATA_ZERO

  NOTE: value in <arg1> is
          0x04    -Command aborted
          0x02    -No media in drive
          0x08    -Media change is requested
          0x20    -Media changed
          0x40    -Write Protected

IOCTL_ENA_REMOVE_NOTIFY
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_DISABLE_REMOVE_NOTIFY
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_DISABLE_OPER
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_ENABLE_ATTRIB_AUTO
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_DISABLE_ATTRIB_AUTO
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_ENABLE_OPER
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_OFFLINE_IMMED
ATA_ZERO          SubCommand          ATA_ZERO            ATA_ZERO
```

```
      (refer to ref1 page no 190)

IOCTL_SMART_READ_DATA   - returns pointer to pointer <ppBuf> of read data
ATA_ZERO            ATA_ZERO          ATA_ZERO            read data

IOCTL_SMART_READ_LOG_SECTOR  - returns pointer to pointer <ppBuf>of read data
ATA_ZERO      no of sector to     log Address    read data
                  be read

IOCTL_SMART_RETURN_STATUS
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_SAVE_ATTRIB
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_SMART_WRITE_LOG_SECTOR
ATA_ZERO          no of to be written     Log Sector address  write data

      NOTE: - <ppBuf> contains pointer to pointer data buffer to be written

IOCTL_CFA_ERASE_SECTORS
ATA_ZERO          sector count        PackedCHS/LBA       ATA_ZERO

IOCTL_CFA_REQUEST_EXTENDED_ERROR_CODE
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_CFA_TRANSLATE_SECTOR - <ppbuf> returns pointer to data pointer.
ATA_ZERO         ATA_ZERO         PackedLBA/CHS       read data

IOCTL_CFA_WRITE_MULTIPLE_WITHOUT_ERASE
ATA_ZERO          sector count        PackedCHS/LBA       write data

      NOTE: -<pbuf> contains pointer to data pointer.

IOCTL_CFA_WRITE_SECTORS_WITHOUT_ERASE
ATA_ZERO          sector count        PackedCHS/LBA       write data
```

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiMaxMDmaModeGet( )

NAME **ichAtapiMaxMDmaModeGet( )** – get the Maximum Multi word DMA mode the drive supports.

SYNOPSIS

```
UINT8 ichAtapiMaxMDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive maximum MDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure The following bits are set for corresponding modes supported.

Bit2	Multi DMA mode 2 and below are supported
Bit1	Multi DMA mode 1 and below are supported
Bit0	Multi DMA mode 0 is supported

RETURNS Maximum Multi word DMA mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiMaxPioModeGet( )

NAME **ichAtapiMaxPioModeGet( )** – get the Maximum PIO mode that drive can support.

SYNOPSIS

```
UINT8 ichAtapiMaxPioModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive maximum PIO mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure

RETURNS maximum PIO mode.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiMaxSDmaModeGet( )

NAME **ichAtapiMaxSDmaModeGet( )** – get the Maximum Single word DMA mode the drive supports

SYNOPSIS

```
UINT8 ichAtapiMaxSDmaModeGet
```

```
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION
This function is used to get drive maximum SDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure

RETURNS
Maximum Single word DMA mode.

ERRNO
Not Available

SEE ALSO
vxbIntelIchStorageShow

ichAtapiMaxUDmaModeGet( )

NAME
ichAtapiMaxUDmaModeGet( ) – get the Maximum Ultra DMA mode the drive can support.

SYNOPSIS
```
UINT8 ichAtapiMaxUDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION
This function is used to get drive maximum UDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. The following bits are set for corresponding modes supported.

Bit4 Ultra DMA mode 4 and below are supported

Bit3 Ultra DMA mode 3 and below are supported

Bit2 Ultra DMA mode 2 and below are supported

Bit1 Ultra DMA mode 1 and below are supported

Bit0 Ultra DMA mode 0 is supported

RETURNS
Maximum Ultra DMA mode.

ERRNO
Not Available

SEE ALSO
vxbIntelIchStorageShow

ichAtapiModelNumberGet( )

NAME

ichAtapiModelNumberGet( ) – get the model number of the drive.

SYNOPSIS

```
char * ichAtapiModelNumberGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function is used to get drive Model Number of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 40 bytes length which contains serial number in ascii.

RETURNS

pointer to the model number.

ERRNO

Not Available

SEE ALSO

vxbIntelIchStorageShow

ichAtapiPktCmd( )

NAME

ichAtapiPktCmd( ) – execute an ATAPI command with error processing

SYNOPSIS

```
UINT8 ichAtapiPktCmd
    (
    ATA_DEV   * pAtapiDev,
    ATAPI_CMD * pComPack
    )
```

DESCRIPTION

This routine executes a single ATAPI command, checks the command completion status and tries to recover if an error encountered during command execution at any stage.

RETURN

SENSE_NO_SENSE if success, or **ERROR** if not successful for any reason.

RETURNS

Not Available

ERRNO

S_ioLib_DEVICE_ERROR

SEE ALSO

vxbIntelIchStorage

ichAtapiPktCmdSend( )

NAME

ichAtapiPktCmdSend( ) – Issue a Packet command.

SYNOPSIS

```
UINT8  ichAtapiPktCmdSend
    (
    ATA_DEV   * pAtapiDev,
    ATAPI_CMD * pComPack
    )
```

DESCRIPTION

This function issues a packet command to specified drive.

See library file description for more details.

RETURN

SENSE_NO_SENSE if success, or **ERROR** if not successful for any reason

RETURNS

Not Available

ERRNO

S_ioLib_DEVICE_ERROR

SEE ALSO

vxbIntelIchStorage

ichAtapiRead10( )

NAME

ichAtapiRead10( ) – read one or more blocks from an ATAPI Device.

SYNOPSIS

```
STATUS ichAtapiRead10
    (
    ATA_DEV   * pAtapiDev,
    UINT32      startBlk,
    UINT32      nBlks,
    UINT32      transferLength,
    char      * pBuf
    )
```

DESCRIPTION

This routine reads one or more blocks from the specified device, starting with the specified block number.

The name of this routine relates to the SFF-8090i (Mt. Fuji), used for DVD-ROM, and indicates that the entire packet command uses 10 bytes, rather than the normal 12.

RETURNS

OK, **ERROR** if the read command didn't succeed.

ERRNO

Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiReadCapacity( )

NAME **ichAtapiReadCapacity( )** – issue a READ CD-ROM CAPACITY command to a ATAPI device

SYNOPSIS

```
STATUS ichAtapiReadCapacity
    (
    ATA_DEV * pAtapiDev
    )
```

DESCRIPTION This routine issues a READ CD-ROM CAPACITY command to a specified ATAPI device.

RETURN **OK**, or **ERROR** if the command fails.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiReadTocPmaAtip( )

NAME **ichAtapiReadTocPmaAtip( )** – issue a READ TOC command to a ATAPI device

SYNOPSIS

```
STATUS ichAtapiReadTocPmaAtip
    (
    ATA_DEV * pAtapiDev,
    UINT32    transferLength,
    char    * resultBuf
    )
```

DESCRIPTION This routine issues a READ TOC command to a specified ATAPI device.

RETURN **OK**, or **ERROR** if the command fails.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiRemovMediaStatusNotifyVerGet( )

NAME **ichAtapiRemovMediaStatusNotifyVerGet( )** – get the Media Stat Notification Version.

SYNOPSIS

```
UINT16 ichAtapiRemovMediaStatusNotifyVerGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the removable media status notification version of the drive.

RETURNS Version Number.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

ichAtapiScan( )

NAME **ichAtapiScan( )** – issue SCAN packet command to ATAPI drive.

SYNOPSIS

```
STATUS ichAtapiScan
    (
    ATA_DEV   * pAtapiDev,
    UINT32      startAddressField,
    int         function
    )
```

DESCRIPTION This function issues SCAN packet command to ATAPI drive. The *function* argument should be 0x00 for fast forward and 0x10 for fast reversed operation.

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiSeek( )

NAME **ichAtapiSeek( )** – issues a SEEK packet command to drive.

SYNOPSIS

```
STATUS ichAtapiSeek
    (
    ATA_DEV * pAtapiDev,
    UINT32    addressLBA
    )
```

DESCRIPTION This function issues a SEEK packet command (not ATA SEEK command) to the specified drive.

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiSetCDSpeed( )

NAME **ichAtapiSetCDSpeed( )** – issue SET CD SPEED packet command to ATAPI drive.

SYNOPSIS

```
STATUS ichAtapiSetCDSpeed
    (
    ATA_DEV   * pAtapiDev,
    int         readDriveSpeed,
    int         writeDriveSpeed
    )
```

DESCRIPTION This function issues SET CD SPEED packet command to ATAPI drive while reading and writing of ATAPI drive(CD-ROM) data. The arguments *readDriveSpeed* and *writeDriveSpeed* are in Kbytes/Second.

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiStartStopUnit( )

NAME **ichAtapiStartStopUnit( )** – Issues START STOP UNIT packet command

SYNOPSIS

```
STATUS ichAtapiStartStopUnit
    (
    ATA_DEV   * pAtapiDev,
    int         arg0
    )
```

DESCRIPTION This function issues a command to drive to MEDIA EJECT and MEDIA LOAD. Argument *arg0* selects to EJECT or LOAD.

To eject media *arg0* should be **EJECT_DISK** To load media *arg0* should be **LOAD_DISK**

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiStopPlayScan( )

NAME **ichAtapiStopPlayScan( )** – issue STOP PLAY/SCAN packet command to ATAPI drive.

SYNOPSIS

```
STATUS ichAtapiStopPlayScan
    (
    ATA_DEV   * pAtapiDev
    )
```

RETURN **OK** or **ERROR**

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiTestUnitRdy( )

NAME **ichAtapiTestUnitRdy( )** – issue a TEST UNIT READY command to a ATAPI drive

SYNOPSIS

```
STATUS ichAtapiTestUnitRdy
    (
    ATA_DEV    * pAtapiDev
    )
```

DESCRIPTION This routine issues a TEST UNIT READY command to a specified ATAPI drive.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorage**

ichAtapiVersionNumberGet( )

NAME **ichAtapiVersionNumberGet( )** – get the ATA/ATAPI version number of the drive.

SYNOPSIS

```
UINT32 ichAtapiVersionNumberGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function will return the ATA/ATAPI version number of the drive. Most significant 16 bits represent the Major Version Number and the Lease significant 16 bits represents the minor Version Number.

Major Version Number

Bit 22	ATA/ATAPI-6
Bit 21	ATA/ATAPI-5
Bit 20	ATA/ATAPI-4
Bit 19	ATA-3

Bit 18	ATA-2

Minor version Number (bit 15 through bit 0)

0001h	Obsolete
0002h	Obsolete
0003h	Obsolete
0004h	ATA-2 published, ANSI X3.279-1996
0005h	ATA-2 X3T10 948D prior to revision 2k
0006h	ATA-3 X3T10 2008D revision 1
0007h	ATA-2 X3T10 948D revision 2k
0008h	ATA-3 X3T10 2008D revision 0
0009h	ATA-2 X3T10 948D revision 3
000Ah	ATA-3 published, ANSI X3.298-199x
000Bh	ATA-3 X3T10 2008D revision 6
000Ch	ATA-3 X3T13 2008D revision 7 and 7a
000Dh	ATA/ATAPI-4 X3T13 1153D revision 6
000Eh	ATA/ATAPI-4 T13 1153D revision 13
000Fh	ATA/ATAPI-4 X3T13 1153D revision 7
0010h	ATA/ATAPI-4 T13 1153D revision 18
0011h	ATA/ATAPI-4 T13 1153D revision 15
0012h	ATA/ATAPI-4 published, ANSI NCITS 317-1998
0013h	Reserved
0014h	ATA/ATAPI-4 T13 1153D revision 14
0015h	ATA/ATAPI-5 T13 1321D revision 1
0016h	Reserved
0017h	ATA/ATAPI-4 T13 1153D revision 17
0018h-FFFFh	Reserved

RETURNS ATA/ATAPI version number.

ERRNO Not Available

SEE ALSO **vxbIntelIchStorageShow**

index( )

NAME **index( )** – find the first occurrence of a character in a string

SYNOPSIS

```
char *index
    (
    FAST const char * s,  /* string in which to find character */
    FAST int          c   /* character to find in string       */
    )
```

DESCRIPTION This routine finds the first occurrence of character *c* in string *s*.

RETURNS A pointer to the located character, or **NULL** if *c* is not found.

ERRNO N/A

SEE ALSO **bLib**, **strchr( )**.

infinity( )

NAME **infinity( )** – return a very large double

SYNOPSIS

```
double infinity (void)
```

DESCRIPTION This routine returns a very large double.

RETURNS The double-precision representation of positive infinity.

ERRNO Not Available

SEE ALSO **mathALib**

infinityf( )

NAME **infinityf( )** – return a very large float

SYNOPSIS

```
float infinityf (void)
```

DESCRIPTION This routine returns a very large float.

RETURNS The single-precision representation of positive infinity.

ERRNO Not Available

SEE ALSO **mathALib**

inflate( )

NAME

inflate( ) – inflate compressed code

SYNOPSIS

```
int inflate
    (
    Byte * src,
    Byte * dest,
    int    nBytes
    )
```

DESCRIPTION

This routine inflates *nBytes* of data starting at address *src*. The inflated code is copied starting at address *dest*. Two sanity checks are performed on the data being decompressed. First, we look for a magic number at the start of the data to verify that it is really a compressed stream. Second, the entire data is optionally checksummed to verify its integrity. By default, the checksum is not verified in order to speed up the booting process. To turn on checksum verification, set the global variable **inflateCksum** to **TRUE** in the BSP.

RETURNS

OK or **ERROR**.

ERRNO

Not Available

SEE ALSO

inflateLib

intCRGet( )

NAME

intCRGet( ) – read the contents of the cause register (MIPS)

SYNOPSIS

```
int intCRGet (void)
```

DESCRIPTION

This routine reads and returns the contents of the MIPS cause register.

RETURNS

The contents of the cause register.

ERRNO

Not Available

SEE ALSO

intArchLib, **intCRSet( )**

intCRSet( )

NAME

intCRSet( ) – write the contents of the cause register (MIPS)

SYNOPSIS

```
void intCRSet
    (
    int value  /* value to write to cause register */
    )
```

DESCRIPTION

This routine writes the contents of the MIPS cause register.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

intArchLib, **intCRGet( )**

intConnect( )

NAME

intConnect( ) – connect a C routine to a hardware interrupt

SYNOPSIS

```
STATUS intConnect
    (
    VOIDFUNCPTR * vector,    /* interrupt vector to attach to     */
    VOIDFUNCPTR   routine,   /* routine to be called              */
    int           parameter  /* parameter to be passed to routine */
    )
```

DESCRIPTION

This routine connects a specified C routine to a specified interrupt vector. The address of *routine* is generally stored at *vector* so that *routine* is called with *parameter* when the interrupt occurs. The routine is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

This routine generally simply calls **intHandlerCreate( )** and **intVecSet( )**. The address of the handler returned by **intHandlerCreate( )** is what actually goes in the interrupt vector.

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE ARM ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to install the handler such that, when the interrupt occurs, *routine* is called with *parameter*.

NOTE X86 Refer to the special x86 routine **intHandlerCreateI86( )**.

NOTE SH The on-chip interrupt controller (INTC) design of SH architecture depends on the processor type, but there are some similarities. The number of external interrupt inputs are limited, so it may necessary to multiplex some interrupt requests. However most of them are auto-vectored, thus have only one vector to an external interrupt input. As a framework to handle this type of multiplexed interrupt, you can use your original intConnect code by hooking it to _func_intConnectHook pointer. If _func_intConnectHook is set, the SH version of **intConnect( )** simply calls the hooked routine with same arguments, then returns the status of hooked routine. A **sysLib** sample is shown below:

```
#include <intLib.h>
#include <iv.h>              /* INUM_INTR_HIGH for SH7750/SH7700 */

#define SYS_INT_TBL_SIZE     (255 - INUM_INTR_HIGH)

typedef struct
    {
    VOIDFUNCPTR routine;     /* routine to be called */
    int         parameter;   /* parameter to be passed */
    } SYS_INT_TBL;

LOCAL SYS_INT_TBL sysIntTbl [SYS_INT_TBL_SIZE]; /* local vector table */

LOCAL int sysInumVirtBase = INUM_INTR_HIGH + 1;

STATUS sysIntConnect
    (
    VOIDFUNCPTR *vec,                /* interrupt vector to attach to     */
    VOIDFUNCPTR routine,     /* routine to be called              */
    int         param                /* parameter to be passed to routine */
    )
    {
    FUNCPTR intDrvRtn;

    if (vec >= INUM_TO_IVEC (0) && vec < INUM_TO_IVEC (sysInumVirtBase))
        {
        /* do regular intConnect() process */

        intDrvRtn = intHandlerCreate ((FUNCPTR)routine, param);

        if (intDrvRtn == NULL)
            return ERROR;

        /* make vector point to synthesized code */

        intVecSet ((FUNCPTR *)vec, (FUNCPTR)intDrvRtn);
        }
    else
```

```
        {
        int index = IVEC_TO_INUM (vec) - sysInumVirtBase;

        if (index < 0 || index >= SYS_INT_TBL_SIZE)
            return ERROR;

        sysIntTbl [index].routine   = routine;
        sysIntTbl [index].parameter = param;
        }

    return OK;
    }

void sysHwInit (void)
    {
    ...
    _func_intConnectHook = (FUNCPTR)sysIntConnect;
    }

LOCAL void sysVmeIntr (void)
    {
    volatile UINT32 vec = *VME_VEC_REGISTER; /* get VME interrupt vector */
    int i = vec - sysInumVirtBase;

    if (i >= 0 && i < SYS_INT_TBL_SIZE && sysIntTbl[i].routine != NULL)
        (*sysIntTbl[i].routine)(sysIntTbl[i].parameter);
    else
        logMsg ("uninitialized VME interrupt: vec = %d\n", vec,0,0,0,0,0);
    }

void sysHwInit2 (void)
    {
    int i;
    ...
    /* initialize VME interrupts dispatch table */

    for (i = 0; i < SYS_INT_TBL_SIZE; i++)
        {
        sysIntTbl[i].routine   = (VOIDFUNCPTR)NULL;
        sysIntTbl[i].parameter = NULL;
        }

    /* connect generic VME interrupts handler */

    intConnect (INT_VEC_VME, sysVmeIntr, NULL);
    ...
    }
```

The used vector numbers of SH processors are limited to certain ranges, depending on the processor type. The **sysInumVirtBase** should be initialized to a value higher than the last used vector number, defined as **INUM_INTR_HIGH**. It is typically safe to set **sysInumVirtBase** to (**INUM_INTR_HIGH** + 1).

The **sysIntConnect( )** routine simply acts as the regular **intConnect( )** if *vector* is smaller than **INUM_TO_IVEC** (sysInumVirtBase), so **sysHwInit2( )** connects a common VME interrupt dispatcher **sysVmeIntr** to the multiplexed interrupt vector. If *vector* is equal to or greater than **INUM_TO_IVEC** (sysInumVirtBase), the **sysIntConnect( )** fills a local vector entry in sysIntTbl[] with an individual VME interrupt handler, in a coordinated manner with **sysVmeIntr**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK**, or **ERROR** if the interrupt handler cannot be built.

ERRNO Not Available

SEE ALSO **intArchLib**, **intHandlerCreate( )**, **intVecSet( )**, **intDisconnect( )**

intContext( )

NAME **intContext( )** – determine if executing in interrupt context

SYNOPSIS

```
BOOL intContext (void)
```

DESCRIPTION This routine returns **TRUE** only if the caller is executing in an interrupt context. If executing in a task context **FALSE** is returned.

SMP CONSIDERATIONS

In a VxWorks SMP system it is possible for the CPUs to be in different contexts (task or interrupt) at the same time. Therefore this routine returns the status for the CPU the caller is executing on.

RETURNS **TRUE** or **FALSE**.

ERRNO N/A

SEE ALSO **intLib**

intCount( )

NAME **intCount( )** – get the current interrupt nesting depth

SYNOPSIS

```
int intCount (void)
```

DESCRIPTION This routine returns the number of interrupts that are currently nested.

SMP CONSIDERATIONS

In a VxWorks SMP system it is possible for the CPUs to be in different contexts (task or interrupt) at the same time. Therefore this routine returns the nested interrupt count for the CPU the caller is executing on.

RETURNS The number of nested interrupts.

ERRNO N/A

SEE ALSO **intLib**

intCpuLock( )

NAME **intCpuLock( )** – lock out interrupts on local CPU

SYNOPSIS

```
int intCpuLock (void)
```

DESCRIPTION This routine disables interrupts on the CPU the calling task or ISR is running on. The returned value is a lock-out key to be used in a subsequent call to **intCpuUnlock( )** to release the lock. Execution of interrupts on other CPUs in the SMP system is not affected by this routine. Because of this behaviour this routine is not a suitable mutual exclusion mechanism unless all tasks and/or ISRs participating in the mutual exclusion scenario have a single CPU affinity to the very same CPU.

Calling this routine on the uniprocessor version of VxWorks is equivalent to calling **intLock( )**.

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.

RETURNS An architecture-dependent lock-out key for the interrupt level prior to the call.

ERRNO Not Available

SEE ALSO **intArchLib**, **intCpuUnlock( )**, **taskCpuLock( )**, **intLockLevelSet( )**

intCpuUnlock( )

NAME **intCpuUnlock( )** – cancel local CPU interrupt lock

SYNOPSIS

```
void intCpuUnlock
    (
    int lockKey  /* lock-out key returned by preceding intCpuLock() */
    )
```

DESCRIPTION This routine removes the lock established using intCpuLock (). It re-enables interrupts on the CPU the calling task or ISR is running on. Calling this routine on the uniprocessor version of VxWorks is equivalent to calling **intUnlock( )**. The parameter *lockKey* is an architecture-dependent lock-out key returned by a preceding **intCpuLock( )** call.

RETURNS N/A

ERRNO Not Available

SEE ALSO **intArchLib**, **intCpuLock( )**, **taskCpuUnlock( )**

intDisable( )

NAME **intDisable( )** – disable corresponding interrupt bits (MIPS, PowerPC, ARM)

SYNOPSIS

```
int intDisable
    (
    int level  /* new interrupt bits (0x0 - 0xff00) */
    )
```

DESCRIPTION On MIPS and PowerPC architectures, this routine disables the corresponding interrupt bits from the present status register.

NOTE ARM ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to disable a particular interrupt level, regardless of the current interrupt mask level.

NOTE MIPS For MIPS, the macros **SR_IBIT1** - **SR_IBIT8** define bits that may be set.

RETURNS **OK** or **ERROR**. (MIPS: The previous contents of the status register).

ERRNO Not Available

SEE ALSO **intArchLib**, **intEnable( )**

intDisconnect( )

NAME **intDisconnect( )** – disconnect a C routine from a hardware interrupt

SYNOPSIS

```
STATUS intDisconnect
    (
    VOIDFUNCPTR * vector,    /* interrupt vector to dettach from  */
    VOIDFUNCPTR   routine,   /* routine to disconnect             */
    int           parameter  /* parameter to be matched           */
    )
```

DESCRIPTION This routine disconnects a specified C routine that has a specified *parameter* from a specified interrupt vector.

The caller of this routine must first disable the source of interrupts before calling this routine.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK**, **ERROR** if the interrupt handler cannot be disconnected.

ERRNO The following are the possible errnos returned when ISR object is supported.

S_intLib_NOT_ISR_CALLABLE
this routine must not be called from an ISR

SEE ALSO **intLib**, **intConnect( )**

intEnable( )

NAME

intEnable( ) – enable corresponding interrupt bits (MIPS, PowerPC, ARM)

SYNOPSIS

```
int intEnable
    (
    int level  /* new interrupt bits (0x00 - 0xff00) */
    )
```

DESCRIPTION

This routine enables the input interrupt bits on the present status register of the MIPS and PowerPC processors.

NOTE ARM

ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to enable the interrupt. For each interrupt level to be used, there must be a call to this routine before it will be allowed to interrupt.

NOTE MIPS

For MIPS, it is strongly advised that the level be a combination of **SR_IBIT1** - **SR_IBIT8**.

RETURNS

OK or **ERROR**. (MIPS: The previous contents of the status register).

ERRNO

Not Available

SEE ALSO

intArchLib, **intDisable( )**

intHandlerCreate( )

NAME

intHandlerCreate( ) – construct an interrupt handler for a C routine (MC680x0, x86, MIPS, SimSolaris)

SYNOPSIS

```
FUNCPTR intHandlerCreate
    (
    FUNCPTR routine,   /* routine to be called               */
    int     parameter  /* parameter to be passed to routine */
    )
```

DESCRIPTION

This routine builds an interrupt handler around the specified C routine. This interrupt handler is then suitable for connecting to a specific vector address with **intVecSet( )**. The interrupt handler is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS
A pointer to the new interrupt handler, or **NULL** if memory is insufficient.

ERRNO
Not Available

SEE ALSO
intArchLib, **intConnect( )**

intHandlerCreateI86( )

NAME
intHandlerCreateI86( ) – construct an interrupt handler for a C routine (x86)

SYNOPSIS

```
FUNCPTR intHandlerCreateI86
    (
    FUNCPTR routine,       /* routine to be called               */
    int     parameter,     /* parameter to be passed to routine */
    FUNCPTR routineBoi,    /* BOI routine to be called           */
    int     parameterBoi,  /* parameter to be passed to routineBoi */
    FUNCPTR routineEoi,    /* EOI routine to be called           */
    int     parameterEoi   /* parameter to be passed to routineEoi */
    )
```

DESCRIPTION
This routine builds an interrupt handler around a specified C routine. This interrupt handler is then suitable for connecting to a specific vector address with **intVecSet( )**. The interrupt handler is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

IMPLEMENTATION
This routine builds an interrupt handler of the following form in allocated memory:

```
00  e8 kk kk kk kk          call    _intEnt          * tell kernel
05  50                      pushl   %eax             * save regs
06  52                      pushl   %edx
07  51                      pushl   %ecx
08  68 pp pp pp pp          pushl   $_parameterBoi   * push BOI param
13  e8 rr rr rr rr          call    _routineBoi      * call BOI routine
18  68 pp pp pp pp          pushl   $_parameter      * push param
23  e8 rr rr rr rr          call    _routine         * call C routine
28  68 pp pp pp pp          pushl   $_parameterEoi   * push EOI param
33  e8 rr rr rr rr          call    _routineEoi      * call EOI routine
38  83 c4 0c                        addl    $12, %esp       * pop param
41  59                      popl    %ecx             * restore regs
42  5a                      popl    %edx
43  58                      popl    %eax
```

```
44  e9 kk kk kk kk          jmp     _intExit        * exit via kernel
```

Third and fourth parameter of **intHandlerCreateI86( )** are the BOI routine address and its parameter that are inserted into the code as "routineBoi" and "parameterBoi". Fifth and sixth parameter of **intHandlerCreateI86( )** are the EOI routine address and its parameter that are inserted into the code as "routineEoi" and "parameterEoi". The BOI routine detects if this interrupt is stray/spurious/phantom by interrogating the interrupt controller, and returns from the interrupt if it is. The EOI routine issues End Of Interrupt signal to the interrupt controller, if it is required by the controller. Each interrupt controller has its own BOI and EOI routine. They are located in the BSP, and their address and parameter are taken by the intEoiGet function (set to **sysIntEoiGet( )** in the BSP). The Tornado 2, and later, BSPs should use the BOI and EOI mechanism with intEoiGet function pointer.

To keep the Tornado 101 BSP backward compatible, the function pointer intEOI is not removed. If intEoiGet is **NULL**, it should be set to the **sysIntEoiGet( )** routine in the BSP, **intHandlerCreate( )** and the intEOI function pointer (set to **sysIntEOI( )** in the Tornado 101 BSP) is used.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

A pointer to the new interrupt handler, or **NULL** if memory is insufficient.

ERRNO

Not Available

SEE ALSO

intArchLib

intLevelSet( )

NAME

intLevelSet( ) – set the interrupt level (MC680X0, x86, ARM, SimSolaris, SimNT and SH)

SYNOPSIS

```
int intLevelSet
    (
    int level  /* new interrupt level mask */
    )
```

DESCRIPTION

This routine changes the interrupt mask in the status register to take on the value specified by *level*. Interrupts are locked out at or below that level. The value of *level* must be in the following range:

MC680x0:	0 - 7
SH:	0 - 15
ARM:	BSP-specific
SimSolaris:	0 - 1

x86:	interrupt controller specific

On x86 systems, there are no interrupt level in the processor and the external interrupt controller manages the interrupt level. Therefore this routine does nothing and returns **OK** always.

NOTE SIMNT This routine does nothing.

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

RETURNS The previous interrupt level.

ERRNO Not Available

SEE ALSO **intArchLib**

intLock( )

NAME **intLock( )** – lock out interrupts

SYNOPSIS

```
int intLock (void)
```

DESCRIPTION This routine disables interrupts. It can be called from either interrupt or task level. The **intLock( )** routine returns an architecture-dependent lock-out key representing the interrupt level prior to the call; this key can be passed to **intUnlock( )** to re-enable interrupts.

For MC680x0, x86, and SH architectures, interrupts are disabled at the level set by **intLockLevelSet( )**. The default lock-out level is the highest interrupt level (MC680x0 = 7, x86 = 1, SH = 15).

For SimSolaris architecture, interrupts are masked. Lock-out level returned is 1 if interrupts were already locked, 0 otherwise.

For SimNT, a windows semaphore is used to lock the interrupts. Lock-out level returned is 1 if interrupts were already locked, 0 otherwise.

For MIPS processors, interrupts are disabled at the master lock-out level; this means no interrupt can occur even if unmasked in the IntMask bits (15-8) of the status register.

For ARM processors, interrupts (IRQs) are disabled by setting the I bit in the CPSR. This means no IRQs can occur.

For PowerPC processors, there is only one interrupt vector. The external interrupt (vector offset 0x500) is disabled when **intLock( )** is called; this means that the processor cannot be interrupted by any external event.

IMPLEMENTATION The lock-out key is implemented differently for different architectures:

MC680x0:	interrupt field mask
MIPS:	status register
x86:	interrupt enable flag (IF) bit from EFLAGS register
PowerPC:	MSR register value
ARM	I bit from the CPSR
SH:	status register
SimSolaris:	1 or 0
SimNT:	1 or 0

WARNINGS Invoking a VxWorks system routine with interrupts locked may result in interrupts being re-enabled for an unspecified period of time. If the called routine blocks, or results in a higher priority task becoming eligible for execution (READY), interrupts will be re-enabled while another task executes, or while the kernel is idle.

To prevent interrupts from being re-enabled for the case where a called routine results in a higher priority task becoming eligible for execution, the **taskLock( )** primitive can be used to disable rescheduling. Note that if a task blocks or suspends, the scheduler will always select the highest priority ready task to execute (or become idle) regardless of whether the task has locked preemption via **taskLock( )**, and thus interrupts will be re-enabled.

The interrupt lock level is an attribute of a task, i.e. it's part of the task context. Thus, if a task disables interrupts and subsequently invokes a VxWorks system routine that causes the calling task to block or cause a higher priority task to be ready, the interrupt lock level will be restored when the task is later rescheduled for execution. For example, in the following code fragment, interrupts will be disabled after returning from the **taskDelay( )** invocation:

```
lockKey = intLock ();

 ... (work with interrupts locked out)

taskDelay (sysClkRateGet() * 10);   /* delay for 10 seconds */

 ... (work with interrupts locked out)
```

Finally, the above descriptions only applies for tasks since ISRs are not permitted to block or suspend, and task rescheduling only occurs when an ISR completes execution.

EXAMPLES

```
lockKey = intLock ();

 ... (work with interrupts locked out)

intUnlock (lockKey);
```

To lock out interrupts and task scheduling as well (see WARNINGS above):

```
    if (taskLock() == OK)
        {
        lockKey = intLock ();

        ... (critical section)

        intUnlock (lockKey);
        taskUnlock();
        }
     else
        {
        ... (error message or recovery attempt)
        }
```

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Refer to the VxWorks SMP Migration Guide for suitable alternatives.

RETURNS An architecture-dependent lock-out key for the interrupt level prior to the call.

ERRNO Not Available

SEE ALSO **intArchLib**, **intUnlock( )**, **taskLock( )**, **intLockLevelSet( )**

intLockLevelGet( )

NAME **intLockLevelGet( )** – get the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT)

SYNOPSIS `int intLockLevelGet (void)`

DESCRIPTION This routine returns the current interrupt lock-out level, which is set by **intLockLevelSet( )** and stored in the globally accessible variable **intLockMask**. This is the interrupt level currently masked when interrupts are locked out by **intLock( )**. The default lock-out level (MC680x0 = 7, x86 = 1, SH = 15) is initially set by **kernelInit( )** when VxWorks is initialized.

NOTE SIMNT This routine does nothing.

RETURNS The interrupt level currently stored in the interrupt lock-out mask. (ARM = **ERROR** always)

ERRNO Not Available

SEE ALSO **intArchLib**, **intLockLevelSet( )**

intLockLevelSet( )

NAME

intLockLevelSet( ) – set the current interrupt lock-out level (MC680x0, x86, ARM, SH, SimSolaris, SimNT)

SYNOPSIS

```
void intLockLevelSet
    (
    int newLevel  /* new interrupt level */
    )
```

DESCRIPTION

This routine sets the current interrupt lock-out level and stores it in the globally accessible variable **intLockMask**. The specified interrupt level is masked when interrupts are locked by **intLock( )**. The default lock-out level (MC680x0 = 7, x86 = 1, SH = 15) is initially set by **kernelInit( )** when VxWorks is initialized.

NOTE SIMSOLARIS, SIMNT

This routine does nothing.

NOTE ARM

On the ARM, this call establishes the interrupt level to be set when **intLock( )** is called.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

intArchLib, **intLockLevelGet( )**, **intLock( )**, **taskLock( )**

intSRGet( )

NAME

intSRGet( ) – read the contents of the status register (MIPS)

SYNOPSIS

```
int intSRGet (void)
```

DESCRIPTION

This routine reads and returns the contents of the MIPS status register.

RETURNS

The previous contents of the status register.

ERRNO

Not Available

SEE ALSO

intArchLib, **intSRSet( )**

intSRSet( )

NAME
intSRSet( ) – update the contents of the status register (MIPS)

SYNOPSIS
```
int intSRSet
    (
    int value  /* value to write to status register */
    )
```

DESCRIPTION
This routine updates and returns the previous contents of the MIPS status register.

RETURNS
The previous contents of the status register.

ERRNO
Not Available

SEE ALSO
intArchLib, **intSRGet( )**

intStackEnable( )

NAME
intStackEnable( ) – enable or disable the interrupt stack usage (x86)

SYNOPSIS
```
STATUS intStackEnable
    (
    BOOL enable  /* TRUE to enable, FALSE to disable */
    )
```

DESCRIPTION
This routine enables or disables the interrupt stack usage and is only callable from the task level. An Error is returned for any other calling context. The interrupt stack usage is disabled in the default configuration for the backward compatibility. Routines that manipulate the interrupt stack, are located in the file **i86/windALib.s**. These routines include **intStackEnable( )**, **intEnt( )** and **intExit( )**.

RETURNS
OK, or **ERROR** if it is not in the task level.

ERRNO
Not Available

SEE ALSO
intArchLib

intUninitVecSet( )

NAME **intUninitVecSet( )** – set the uninitialized vector handler (ARM)

SYNOPSIS

```
void intUninitVecSet
    (
    VOIDFUNCPTR routine  /* ptr to user routine */
    )
```

DESCRIPTION This routine installs a handler for the uninitialized vectors to be called when any uninitialised vector is entered.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **intArchLib**

intUnlock( )

NAME **intUnlock( )** – cancel interrupt locks

SYNOPSIS

```
void intUnlock
    (
    int lockKey  /* lock-out key returned by preceding intLock() */
    )
```

DESCRIPTION This routine re-enables interrupts that have been disabled by **intLock( )**. The parameter *lockKey* is an architecture-dependent lock-out key returned by a preceding **intLock( )** call.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Refer to the VxWorks SMP Migration Guide for suitable alternatives.

RETURNS N/A

ERRNO Not Available

SEE ALSO **intArchLib**, **intLock( )**

intVecBaseGet( )

NAME

intVecBaseGet( ) – get the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT)

SYNOPSIS

```
FUNCPTR *intVecBaseGet (void)
```

DESCRIPTION

This routine returns the current vector base address, which is set with **intVecBaseSet( )**.

RETURNS

The current vector base address (MIPS = 0 always, ARM = 0 always, SimSolaris = 0 always and SimNT = 0 always).

ERRNO

Not Available

SEE ALSO

intArchLib, **intVecBaseSet( )**

intVecBaseSet( )

NAME

intVecBaseSet( ) – set the vector (trap) base address (MC680x0, x86, MIPS, ARM, SimSolaris, SimNT)

SYNOPSIS

```
void intVecBaseSet
    (
    FUNCPTR * baseAddr  /* new vector (trap) base address */
    )
```

DESCRIPTION

This routine sets the vector (trap) base address. The CPU's vector base register is set to the specified value, and subsequent calls to **intVecGet( )** or **intVecSet( )** will use this base address. The vector base address is initially 0, until modified by calls to this routine.

NOTE 68000

The 68000 has no vector base register; thus, this routine is a no-op for 68000 systems.

NOTE MIPS

The MIPS processors have no vector base register; thus this routine is a no-op for this architecture.

NOTE SH77XX

This routine sets *baseAddr* to vbr, then loads an interrupt dispatch code to (vbr + 0x600). When SH77XX processor accepts an interrupt request, it sets an exception code to INTEVT register and jumps to (vbr + 0x600). Thus this dispatch code is commonly used for all interrupts' handling.

The exception codes are 12bits width, and interleaved by 0x20. VxWorks for SH77XX locates a vector table at (vbr + 0x800), and defines the vector offsets as (exception codes / 8). This vector table is commonly used by all interrupts, exceptions, and software traps.

All SH77XX processors have INTEVT register at address 0xffffffd8. The SH7707 processor has yet another INTEVT2 register at address 0x04000000, to identify its enhanced interrupt sources. The dispatch code obtains the address of INTEVT register from a global constant **intEvtAdrs**. The constant is defined in **sysLib**, thus the selection of INTEVT/INTEVT2 is configurable at BSP level. The **intEvtAdrs** is loaded to (vbr + 4) by **intVecBaseSet( )**.

After fetching the exception code, the interrupt dispatch code applies a new interrupt mask to the status register, and jumps to an individual interrupt handler. The new interrupt mask is taken from **intPrioTable[]**, which is defined in **sysALib**. The **intPrioTable[]** is loaded to (vbr + 0xc00) by **intVecBaseSet( )**.

NOTE ARM
The ARM processors have no vector base register; thus this routine is a no-op for this architecture.

NOTE SIMSOLARIS, SIMNT
This routine does nothing.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
intArchLib, **intVecBaseGet( )**, **intVecGet( )**, **intVecSet( )**

intVecGet( )

NAME
intVecGet( ) – get an interrupt vector (MC680x0, x86, MIPS, SH, SimSolaris, SimNT)

SYNOPSIS
```
FUNCPTR intVecGet
    (
    FUNCPTR * vector  /* vector offset */
    )
```

DESCRIPTION
This routine returns a pointer to the exception/interrupt handler attached to a specified vector. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at:

MC680x0:	0
MIPS:	**excBsrTbl** in **excArchLib**
x86:	0
SH702x/SH703x/SH704x/SH76xx:	**excBsrTbl** in **excArchLib**

SH77xx:	vbr + 0x800
SimSolaris:	0

However, the vector table may be set to start at any address with **intVecBaseSet( )** (on CPUs for which it is available).

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE SIMNT This routine does nothing and always returns 0.

RETURNS A pointer to the exception/interrupt handler attached to the specified vector.

ERRNO Not Available

SEE ALSO **intArchLib**, **intVecSet( )**, **intVecBaseSet( )**

intVecGet2( )

NAME **intVecGet2( )** – get a CPU vector, gate type(int/trap), and gate selector (x86)

SYNOPSIS

```
void intVecGet2
    (
    FUNCPTR * vector,       /* vector offset              */
    FUNCPTR * pFunction,    /* address to place in vector */
    int *     pIdtGate,     /* IDT_TRAP_GATE or IDT_INT_GATE */
    int *     pIdtSelector  /* sysCsExc or sysCsInt */
    )
```

DESCRIPTION This routine gets a pointer to the exception/interrupt handler attached to a specified vector, the type of the gate, the selector of the gate. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at address 0. However, the vector table may be set to start at any address with **intVecBaseSet( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **intArchLib**, **intVecBaseSet( )**, **intVecGet( )**, **intVecSet( )**, **intVecSet2( )**

intVecSet( )

NAME

intVecSet( ) – set a CPU vector (trap) (MC680x0, x86, MIPS, SH, SimSolaris, SimNT)

SYNOPSIS

```
void intVecSet
    (
    FUNCPTR * vector,   /* vector offset              */
    FUNCPTR   function  /* address to place in vector */
    )
```

DESCRIPTION

This routine attaches an exception/interrupt/trap handler to a specified vector. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at:

MC680x0:	0
MIPS:	**excBsrTbl** in **excArchLib**
x86:	0
SH702x/SH703x/SH704x/SH76xx:	**excBsrTbl** in **excArchLib**
SH77xx:	vbr + 0x800
SimSolaris:	0

However, the vector table may be set to start at any address with **intVecBaseSet( )** (on CPUs for which it is available). The vector table is set up in **usrInit( )**.

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

The **intVecSet( )** routine puts this generated code into the trap table entry corresponding to *vector*.

Window overflow and window underflow are sacred to the kernel and may not be pre-empted. They are written here only to track changing trap base registers (TBRs). With the "branch anywhere" scheme (as opposed to the branch PC-relative +/-8 megabytes) the first instruction in the vector table must not be a change of flow control nor affect any critical registers. The JMPL that replaces the BA will always execute the next vector's first instruction.

NOTE MIPS

On MIPS CPUs the vector table is set up statically in software.

NOTE SH77XX

The specified interrupt handler *function* has to coordinate with an interrupt stack frame which is specially designed for SH77XX version of VxWorks:

```
   [ task's stack ]       [ interrupt stack ]

     |  xxx  | high address
     |  yyy  |                +-------+
     |__zzz__|<---------------|task'sp|  0
     |       |                |INTEVT | -4
     |       | low address    |  ssr  | -8
```

```
                |_ spc _| -12 <- sp (non-nested interrupt)
                :       :
                :       :
                :_______:
                 |INTEVT |  0
                 |  ssr  | -4
                 |_ spc _| -8  <- sp (nested interrupt)
                 |       |
```

This interrupt stack frame is formed by a common interrupt dispatch code which is loaded at (vbr + 0x600). You usually do not have to pay any attention to this stack frame, since **intConnect( )** automatically appends an appropriate stack manipulation code to your interrupt service routine. The **intConnect( )** assumes that your interrupt service routine (ISR) is written in C, thus it also wraps your ISR in minimal register save/restore codes. However if you need a very fast response time to a particular interrupt request, you might want to skip this register save/restore sequence by directly attaching your ISR to the corresponding vector table entry using **intVecSet( )**. Note that this technique is only applicable to an interrupt service with NO VxWorks system call. For example it is not allowed to use **semGive( )** or **logMsg( )** in the interrupt service routine which is directly attached to vector table by **intVecSet( )**. To facilitate the direct usage of **intVecSet( )** by user, a special entry point to exit an interrupt context is provided within the SH77XX version of VxWorks kernel. This entry point is located at address (vbr + intRte1W), here the intRte1W is a global symbol for the vbr offset of the entry point in 16 bit length. This entry point **intRte1** assumes that the current register bank is 0 (SR.RB == 0), and r1 and r0 are still saved on the interrupt stack, and it also requires 0x70000000 in r0. Then **intRte1** properly cleans up the interrupt stack and executes *rte* instruction to return to the previous interrupt or task context. The following code is an example of **intRte1** usage. Here the corresponding intPrioTable[] entry is assumed to be 0x400000X0, namely MD=1, RB=0, BL=0 at the beginning of **usrIsr1**.

```
     .text
     .align  2
     .global _usrIsr1
     .type   _usrIsr1,@function
     .extern _usrRtn
     .extern _intRte1W
                              /* intPrioTable[] sets SR to 0x400000X0 */
_usrIsr1:
     mov.l   r0,@-sp          /* must save r0 first (BANK0) */
     mov.l   r1,@-sp          /* must save r1 second (BANK0) */

    mov.l   r2,@-sp          /* save rest of volatile registers (BANK0) */
    mov.l   r3,@-sp
    mov.l   r4,@-sp
    mov.l   r5,@-sp
    mov.l   r6,@-sp
    mov.l   r7,@-sp
    sts.l   pr,@-sp
    sts.l   mach,@-sp
    sts.l   macl,@-sp

    mov.l   UsrRtn,r0
    jsr     @r0             /* call user's C routine */
    nop                     /* (delay slot) */

    lds.l   @sp+,macl       /* restore volatile registers (BANK0) */
    lds.l   @sp+,mach
    lds.l   @sp+,pr
    mov.l   @sp+,r7
    mov.l   @sp+,r6
    mov.l   @sp+,r5
    mov.l   @sp+,r4
    mov.l   @sp+,r3
    mov.l   @sp+,r2
                            /* intRte1 restores r1 and r0 */
    mov.l   IntRte1W,r1
    mov.w   @r1,r0
    stc     vbr,r1
    add     r0,r1
    mov.l   IntRteSR,r0     /* r0: 0x70000000 */
    jmp     @r1             /* let intRte1 clean up stack, then rte */
    nop                     /* (delay slot) */

            .align  2
UsrRtn:     .long   _usrRtn         /* user's C routine */
IntRteSR:   .long   0x70000000      /* MD=1, RB=1, BL=1 */
IntRte1W:   .long   _intRte1W
```

The **intRte1** sets r0 to status register (SR: 0x70000000), to safely restore SPC/SSR and to clean up the interrupt stack. Note that TLB mishit exception immediately reboots CPU while SR.BL=1. To avoid this fatal condition, VxWorks loads the **intRte1** code and the interrupt stack to a physical address space (P1) where no TLB mishit happens.

Furthermore, there is another special entry point called **intRte2** at an address (vbr + intRte2W). The **intRte2** assumes that SR is already set to 0x70000000 (MD: 1, RB: 1, BL: 1), then it does not restore r1 and r0. While SR value is 0x70000000, you may use r0,r1,r2,r3 in BANK1 as volatile registers. The rest of BANK1 registers (r4,r5,r6,r7) are non-volatile, so if you need to use them then you have to preserve their original values by saving/restoring them on the interrupt stack. So, if you need the ultimate interrupt response time, you may set the corresponding intPrioTable[] entry to **NULL** and manage your interrupt service only with r0,r1,r2,r3 in BANK1 as shown in the next sample code:

```
    .text
    .global  _usrIsr2
    .type    _usrIsr2,@function
    .extern  _usrIntCnt     /* interrupt counter */
    .extern  _intRte2W
    .extern  _vxShP1TextBase
    .align   2
                            /* MD=1, RB=1, BL=1, since SR is not */
                            /* substituted from intPrioTable[].  */
_usrIsr2:
    mov.l   UsrIntAck,r1
    mov     #0x1,r0
    mov.b   r0,@r1          /* acknowledge interrupt */
    mov.l   UsrIntCnt,r1
    mov.l   X1FFFFFFF,r2
    mov.l   UsrP1Base,r3
    mov.l   @r3,r3          /* r3: SH_P1_TEXT_BASE */
    and     r2,r1
    or      r3,r1           /* r1: _usrIntCnt address in P1 */
    mov.l   @r1,r0
    add     #1,r0
    mov.l   r0,@r1          /* increment counter */

    mov.l   IntRte2W,r1
    and     r2,r1
    or      r3,r1           /* r1: intRte2W address in P1 */
    mov.w   @r1,r0
    stc     vbr,r1
    add     r1,r0
    jmp     @r0             /* let intRte2 clean up stack, then rte */
    nop                     /* (delay slot) */

            .align  2
UsrIntAck:  .long   0xa0001234      /* interrupt acknowledge register */
UsrIntCnt:  .long   _usrIntCnt
IntRte2W:   .long   _intRte2W
X1FFFFFFF:  .long   0x1fffffff
UsrP1Base:  .long   _vxShP1TextBase
```

Note that the entire interrupt service is executed under SR.BL=1 in this sample code. It means that any access to virtual address space may reboot CPU, since TLB mishit exception is blocked. Therefore **usrIsr2** has to access **usrIntCnt** and **intRte2W** from P1 region. Also **usrIsr2** itself has to be executed on P1 region, and it can be done by relocating the address of **usrIsr2** to P1 as shown below:

```
IMPORT void usrIsr2 (void);

intVecSet (vector, (FUNCPTR) usrIsr2);
```

In conclusion, you have to guarantee that the entire ISR does not access to any virtual address space if you set the corresponding intPrioTable[] entry to **NULL**.

NOTE SIMNT This routine does nothing.

RETURNS N/A

ERRNO Not Available

SEE ALSO **intArchLib**, **intVecBaseSet( )**, **intVecGet( )**

intVecSet2( )

NAME

intVecSet2( ) – set a CPU vector, gate type(int/trap), and selector (x86)

SYNOPSIS

```
void intVecSet2
    (
    FUNCPTR * vector,      /* vector offset              */
    FUNCPTR   function,    /* address to place in vector */
    int       idtGate,     /* IDT_TRAP_GATE or IDT_INT_GATE */
    int       idtSelector  /* sysCsExc or sysCsInt */
    )
```

DESCRIPTION

This routine attaches an exception handler to a specified vector, with the type of the gate and the selector of the gate. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at address 0. However, the vector table may be set to start at any address with **intVecBaseSet( )**. The vector table is set up in **usrInit( )**.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

intArchLib, **intVecBaseSet( )**, **intVecGet( )**, **intVecSet( )**, **intVecGet2( )**

intVecTableWriteProtect( )

NAME

intVecTableWriteProtect( ) – write-protect exception vector table (MC680x0, x86, ARM, SimSolaris, SimNT)

SYNOPSIS

```
STATUS intVecTableWriteProtect (void)
```

DESCRIPTION

If the unbundled Memory Management Unit (MMU) support package (VxVMI) is present, this routine write-protects the exception vector table to protect it from being accidentally corrupted.

Note that other data structures contained in the page will also be write-protected. In the default VxWorks configuration, the exception vector table is located at location 0 in memory. Write-protecting this affects the backplane anchor, boot configuration information, and potentially the text segment (assuming the default text location of 0x1000.) All code that manipulates these structures has been modified to write-enable memory for the duration of the operation. If you select a different address for the exception vector table, be sure it resides in a page separate from other writable data structures.

NOTE SIMSOLARIS, SIMNT

This routine always returns **ERROR** on simulators.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK**, or **ERROR** if memory cannot be write-protected.

ERRNO **S_intLib_VEC_TABLE_WP_UNAVAILABLE**

SEE ALSO **intArchLib**

ioGlobalStdGet( )

NAME **ioGlobalStdGet( )** – get the file descriptor for global input/output/error

SYNOPSIS

```
int ioGlobalStdGet
    (
    int stdFd  /* std input (0), output (1), or error (2) */
    )
```

DESCRIPTION This routine returns the current underlying file descriptor for global standard input, output, and error.

RETURNS The underlying global file descriptor, or **ERROR** if *stdFd* is not 0, 1, or 2.

ERRNO N/A.

SEE ALSO **ioLib**, **ioGlobalStdSet( )**, **ioTaskStdGet( )**

ioGlobalStdSet( )

NAME **ioGlobalStdSet( )** – set file descriptor for global input/output/error

SYNOPSIS

```
STATUS ioGlobalStdSet
    (
    int stdFd,  /* std input (0), output (1), or error (2) */
    int newFd   /* new underlying file descriptor          */
    )
```

DESCRIPTION

This routine changes the assignment of a specified global standard file descriptor *stdFd* (0, 1, or, 2) to the specified underlying file descriptor *newFd*. *newFd* should be a file descriptor open to the desired device or file. All tasks will use this new assignment when doing I/O to *stdFd*, unless they have specified a task-specific standard file descriptor (see **ioTaskStdSet( )**). If *stdFd* is not 0, 1, or 2, this routine has no effect.

RETURNS

OK, or **ERROR** if input data is not valid.

ERRNO

EBADF
The newFd does not represent a valid open file.

EINVAL
The stdFd value is not valid.

SEE ALSO

ioLib, **ioGlobalStdGet( )**, **ioTaskStdSet( )**

ioHelp( )

NAME

ioHelp( ) – print a synopsis of I/O utility functions

SYNOPSIS

```
void ioHelp (void)
```

DESCRIPTION

This function prints out synopsis for the I/O and File System utility functions.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

usrFsLib, the VxWorks programmer guides.

ioTaskStdGet( )

NAME

ioTaskStdGet( ) – get the file descriptor for task standard input/output/error

SYNOPSIS

```
int ioTaskStdGet
    (
    int taskId,  /* ID of desired task (0 = self)           */
    int stdFd    /* std input (0), output (1), or error (2) */
    )
```

DESCRIPTION This routine returns the current underlying file descriptor for task-specific standard input, output, and error.

RETURNS The underlying file descriptor, or **ERROR** if *stdFd* is not 0, 1, or 2, or the routine is called at interrupt level.

ERRNO N/A.

SEE ALSO **ioLib**, **ioGlobalStdGet( )**, **ioTaskStdSet( )**

ioTaskStdSet( )

NAME **ioTaskStdSet( )** – set the file descriptor for task standard input/output/error

SYNOPSIS

```
STATUS ioTaskStdSet
    (
    int taskId,  /* task whose std fd is to be set (0 = self) */
    int stdFd,   /* std input (0), output (1), or error (2)   */
    int newFd    /* new underlying file descriptor            */
    )
```

DESCRIPTION This routine changes the assignment of a specified task-specific standard file descriptor *stdFd* (0, 1, or, 2) to the specified underlying file descriptor*newFd*. *newFd* should be a file descriptor open to the desired device or file. The calling task will use this new assignment when doing I/O to *stdFd*, instead of the system-wide global assignment which is used by default. If *stdFd* is not 0, 1, or 2, this routine has no effect.

NOTE This routine has no effect if it is called at interrupt level.

RETURNS **OK**, or **ERROR** if input data is not valid.

ERRNO

EBADF
The newFd does not represent a valid open file.

EINVAL
The stdFd or taskId values are not valid.

SEE ALSO **ioLib**, **ioGlobalStdGet( )**, **ioTaskStdGet( )**

ioctl( )

NAME

ioctl( ) – perform an I/O control function

SYNOPSIS

```
int ioctl
    (
    int fd,        /* file descriptor    */
    int function,  /* function code      */
    ...
    )
```

DESCRIPTION

This routine performs an I/O control function on a device. The control functions used by VxWorks device drivers are defined in the header file **ioLib.h**. Most requests are passed on to the driver for handling. Since the availability of **ioctl( )** functions is driver-specific, these functions are discussed separately in **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, and **rawFsLib**.

The following example renames the file or directory to the string "newname":

```
    ioctl (fd, FIORENAME, "newname");
```

Note that the function FIOGETNAME is handled by the I/O interface level and is not passed on to the device driver itself. Thus this function code value should not be used by customer-written drivers. This ioctl code is convenient for checking the validity of any *fd* number. The following example shows how to quickly validate a given *fd* number as being valid.

```
   if (ioctl (fd, FIOGETNAME, NULL) == ERROR)
       {
       /* fd is not valid */
       }
```

RETURNS

The return value of the driver, or **ERROR** if the file descriptor does not exist.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

ENXIO
Device and its driver are removed. **close( )** should be called to release this file descriptor.

Other
Other errors reported by device driver.

SEE ALSO

ioLib, **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rawFsLib**, the VxWorks programmer guides.

iosDevAdd( )

NAME

iosDevAdd( ) – add a device to the kernel I/O system

SYNOPSIS

```
STATUS iosDevAdd
    (
    DEV_HDR    *pDevHdr,  /* pointer to device's structure */
    const char *name,     /* name of device */
    int        drvnum     /* no. of servicing driver, */
                          /* returned by iosDrvInstall() */
    )
```

DESCRIPTION

This routine adds a device to the I/O system device list, making the device available for subsequent **open( )** and **creat( )** calls.

The parameter *pDevHdr* is a pointer to a device header, **DEV_HDR** (defined in **ioLib.h**), which is used as the node in the device list. Usually this is the first item in a larger device structure for the specific device type. The parameters *name* and *drvnum* are entered in *pDevHdr*.

RETURNS

OK, or **ERROR** if there is already a device with the specified name.

ERRNO

S_iosLib_DUPLICATE_DEVICE_NAME (**EINVAL**)
Device name already in use.

EINVAL
invalid arguments

SEE ALSO

iosLib

iosDevDelDrv( )

NAME

iosDevDelDrv( ) – invoke device delete driver if reference counter reaches 0.

SYNOPSIS

```
int iosDevDelDrv
    (
    DEV_HDR *pDevHdr  /* pointer to device's structure */
    )
```

DESCRIPTION

This routine invokes device delete driver if reference counter reaches 0.

If the device was never added to the device list, unpredictable results may occur.

RETURNS

DELETE_DONE
Device deleted successfully. Driver called if being set.

REFCNT_NOT_ZERO
Device driver reference counter is not zero, device delete driver invocation is delayed until the reference counter reaches zero. The device entry is deleted and all file descriptors open on the device are invalidated in this case.

DELETE_ERROR
Error encountered in device delete.

ERRNO

N/A.

SEE ALSO

iosLib

iosDevDelete( )

NAME

iosDevDelete( ) – delete a device from the kernel I/O system

SYNOPSIS

```
int iosDevDelete
    (
    DEV_HDR *pDevHdr  /* pointer to device's structure */
    )
```

DESCRIPTION

This routine deletes a device from the I/O system device list, making it unavailable to subsequent IO accesses. The driver of the device is intact. **iosDrvRemove( )** will do the same as this function and remove the driver in addition.

All file descriptors open on the device are invalidated which will fail all subsequent use other than **close( )** on them. They are held, even invalidated, continuously until the holding application closes them and thus releases the resource.

If a device delete callback function is installed by **iosDevDelCallback( )**, it will be called if the device driver reference counter is zero. Otherwise the callback invocation is delayed.

If the device was never added to the device list, unpredictable results may occur.

RETURNS

OK (**DELETE_DONE**)
Device deleted successfully.

REFCNT_NOT_ZERO
Device deleted successfully. This code is returned only when the device delete callback function is installed by **iosDevDelCallback( )**. When the device driver reference counter is not zero, the callback invocation is delayed until the reference counter reaches zero when the last device driver returns. However the device entry is already deleted and all file descriptors open on the device are invalidated in this case.

ERROR (**DELETE_ERROR**)
Error encountered in device delete.

ERRNO
EINVAL
Invalid argument. Device already deleted, or not installed, etc.

SEE ALSO
iosLib

iosDevFind( )

NAME
iosDevFind( ) – find an I/O device in the kernel device list

SYNOPSIS

```
DEV_HDR *iosDevFind
    (
    const char *name,          /* name of the device */
    const char *(*pNameTail)  /* where to put ptr to tail of name */
    )
```

DESCRIPTION
This routine searches the device list for a device whose name matches the first portion of *name*. If a device is found, **iosDevFind( )** sets the character pointer pointed to by *pNameTail* to point to the first character in *name*, following the portion which matched the device name. It then returns a pointer to the device. If the routine fails, it returns a pointer to the default device (that is, the device where the current working directory is mounted) and sets *pNameTail* to point to the beginning of *name*. If there is no default device, **iosDevFind( )** returns **NULL**.

RETURNS
A pointer to the device header, or **NULL** if the device is not found.

ERRNO
EINVAL
Invalid arguments.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No device found.

SEE ALSO
iosLib

iosDevShow( )

NAME

iosDevShow( ) – display the list of devices in the system

SYNOPSIS

```
void iosDevShow (void)
```

DESCRIPTION

This routine displays a list of all devices in the device list.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

iosShow, **devs( )**, **windsh**, the VxWorks programmer guides, and the IDE and host tools guides.

iosDrvInstall( )

NAME

iosDrvInstall( ) – install a kernel I/O driver

SYNOPSIS

```
int iosDrvInstall
    (
    FUNCPTR pCreate,  /* pointer to driver create function */
    FUNCPTR pRemove,  /* pointer to driver remove function */
    FUNCPTR pOpen,    /* pointer to driver open function */
    FUNCPTR pClose,   /* pointer to driver close function */
    FUNCPTR pRead,    /* pointer to driver read function */
    FUNCPTR pWrite,   /* pointer to driver write function */
    FUNCPTR pIoctl    /* pointer to driver ioctl function */
    )
```

DESCRIPTION

This routine should be called once by each I/O driver. It hooks up the various I/O service calls to the driver service routines, assigns the driver a number, and adds the driver to the driver table.

RETURNS

The driver number of the new driver, or **ERROR** if there is no room for the driver.

ERRNO

S_iosLib_DRIVER_GLUT (**ENOMEM**)
No memory available for data structures.

SEE ALSO

iosLib

iosDrvRemove( )

NAME

iosDrvRemove( ) – remove a kernel I/O driver

SYNOPSIS

```
STATUS iosDrvRemove
    (
    int  drvnum,     /* no. of driver to remove,             */
                     /* returned by iosDrvInstall()          */
    BOOL forceClose  /* if TRUE, force closure of open files */
    )
```

DESCRIPTION

This routine removes an I/O driver (added by **iosDrvInstall( )**) from the driver table and all device header entries which access the driver.

The parameter *drvnum* is an indicator of driver to be removed that is the number returned by **iosDrvInstall( )**. If *forceClose* is true, all open file descriptors on this device will be closed even they are not closed by applications. This is not recommended. If *forceClose* is not true, file descriptors will be invalidated which will fail all IO other than **close( )** on them. This is the graceful behavior in the driver removal case.

If a device delete callback function is installed to a device by **iosDevDelCallback( )**, it will be called if the device driver reference counter is zero when the device entry is deleted. Otherwise the callback invocation is delayed.

RETURNS

OK
Driver and Device entries removed successfully.

REFCNT_NOT_ZERO
Driver and Device entries are removed successfully as returning **OK** case. This code is returned only when the device delete callback function is installed by **iosDevDelCallback( )**. When the device driver reference counter is not zero, the callback invocation is delayed until the reference counter reaches zero when the last device driver returns.

ERROR
Error encountered in driver & device delete.

ERRNO

EINVAL
invalid arguments

SEE ALSO

iosLib, **iosDrvInstall( )**, **iosDevDelete( )**

iosDrvShow( )

NAME **iosDrvShow( )** – display a list of system drivers

SYNOPSIS

```
void iosDrvShow (void)
```

DESCRIPTION This routine displays a list of all drivers in the driver list. It now includes the null driver, which was previously omitted from the list.

RETURNS N/A

ERRNO N/A

SEE ALSO **iosShow**, **windsh**, the VxWorks programmer guides, and the IDE and host tools guides.

iosFdEntryGet( )

NAME **iosFdEntryGet( )** – get an unused **FD_ENTRY** from the pool

SYNOPSIS

```
FD_ENTRY* iosFdEntryGet (void)
```

DESCRIPTION Returns an unused **FD_ENTRY** pointer, or **NULL**.

RETURNS Returns a pointer to the item, or **NULL** if none is available.

ERRNO **ENFILE**
Too many open files, no internal structures available.

SEE ALSO **iosLib**

iosFdEntryReturn( )

NAME **iosFdEntryReturn( )** – return an **FD_ENTRY** to the pool

SYNOPSIS

```
STATUS iosFdEntryReturn
    (
    FD_ENTRY * pFdEntry  /* entry to be returned to pool */
    )
```

DESCRIPTION The **FD_ENTRY** argument is returned to the **FD_ENTRY** pool.

RETURNS **OK**, always.

ERRNO N/A.

SEE ALSO **iosLib**

iosFdMaxFiles( )

NAME **iosFdMaxFiles( )** – return maximum files for current RTP

SYNOPSIS

```
size_t iosFdMaxFiles (void)
```

DESCRIPTION Returns the maximum number of open files for the current RTP.

RETURNS The maximum number of files for the current RTP. The highest valid file descriptor number will be one less than this value.

ERRNO N/A.

SEE ALSO **iosLib**

iosFdShow( )

NAME **iosFdShow( )** – display a list of file descriptor names in the system

SYNOPSIS

```
void iosFdShow (void)
```

DESCRIPTION This routine displays a list of all file descriptors in the system.

RETURNS N/A

ERRNO N/A

SEE ALSO **iosShow**, **ioctl( )**, **windsh**, the VxWorks programmer guides, and the IDE and host tools guides.

iosInit( )

NAME

iosInit( ) – initialize the kernel I/O system

SYNOPSIS

```
STATUS iosInit
    (
    int         max_drivers,  /* maximum number of drivers allowed */
    int         max_files,    /* max number of files allowed open at once */
    const char* nullDevName   /* name of the null device (bit bucket) */
    )
```

DESCRIPTION

This routine initializes the kernel I/O system. It must be called before any other I/O system routine.

RETURNS

OK, or **ERROR** if memory is insufficient.

ERRNO

N/A.

SEE ALSO

iosLib

iosRtpFdShow( )

NAME

iosRtpFdShow( ) – show the per-RTP *fd* table

SYNOPSIS

```
STATUS iosRtpFdShow
    (
    RTP_ID rtpId
    )
```

DESCRIPTION

Primarily a debugging aid, this routine displays the **FD_ENTRY** pointers for all open file descriptors in a specified RTP.

RETURNS

OK, or **ERROR** if the argument is invalid.

ERRNO

N/A

SEE ALSO

iosShow

iosShowInit( )

NAME **iosShowInit( )** – initialize the I/O system show facility

SYNOPSIS
```
void iosShowInit (void)
```

DESCRIPTION This routine links the I/O system show facility into the VxWorks system. It is called automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

RETURNS N/A

ERRNO N/A

SEE ALSO **iosShow**

irint( )

NAME **irint( )** – convert a double-precision value to an integer

SYNOPSIS
```
int irint
    (
    double x  /* argument */
    )
```

DESCRIPTION This routine converts a double-precision value *x* to an integer using the selected IEEE rounding direction.

CAVEAT The rounding direction is not pre-selectable and is fixed for round-to-the-nearest.

RETURNS The integer representation of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

irintf( )

NAME

irintf( ) – convert a single-precision value to an integer

SYNOPSIS

```
int irintf
    (
    float x  /* argument */
    )
```

DESCRIPTION

This routine converts a single-precision value *x* to an integer using the selected IEEE rounding direction.

CAVEAT

The rounding direction is not pre-selectable and is fixed as round-to-the-nearest.

RETURNS

The integer representation of *x*.

ERRNO

Not Available

SEE ALSO

mathALib

iround( )

NAME

iround( ) – round a number to the nearest integer

SYNOPSIS

```
int iround
    (
    double x  /* argument */
    )
```

DESCRIPTION

This routine rounds a double-precision value *x* to the nearest integer value.

NOTE

If *x* is spaced evenly between two integers, it returns the even integer.

RETURNS

The integer nearest to *x*.

ERRNO

Not Available

SEE ALSO

mathALib

iroundf( )

NAME **iroundf( )** – round a number to the nearest integer

SYNOPSIS

```
int iroundf
    (
    float x  /* argument */
    )
```

DESCRIPTION This routine rounds a single-precision value *x* to the nearest integer value.

NOTE If *x* is spaced evenly between two integers, the even integer is returned.

RETURNS The integer nearest to *x*.

ERRNO Not Available

SEE ALSO **mathALib**

isatty( )

NAME **isatty( )** – return whether the underlying driver is a *tty* device

SYNOPSIS

```
BOOL isatty
    (
    int fd  /* file descriptor to check */
    )
```

DESCRIPTION This routine simply invokes the **ioctl( )** function FIOISATTY on the specified file descriptor.

RETURNS **TRUE**, or **FALSE** if the driver does not indicate a *tty* device.

ERRNO See **ioctl( )**.

SEE ALSO **ioLib**

isrCreate( )

NAME

isrCreate( ) – create an ISR object

SYNOPSIS

```
ISR_ID isrCreate
    (
    char *  name,         /* name of ISR object            */
    UINT    isrTag,       /* interrupt identifier          */
    FUNCPTR handlerRtn,   /* handler routine               */
    int     parameter,    /* parameter to handler routine */
    UINT    options       /* not used, must be set to 0           */
    )
```

DESCRIPTION

This routine creates an ISR object and initializes it with the values passed as arguments. It is meant to be used by code which connects routines to interrupt vectors by means other than calling **intConnect( )** since that routine implicitly creates an ISR object.

The *name* argument is the actual name of the ISR object. Any null terminated ASCII string is acceptable. The name is merely used for debugging purposes. Should *name* be **NULL**, a name is created using a "isrN" pattern where "N" is a decimal number representing the Nth ISR object for which the system had to choose a name at creation. Numbering starts at N = 1. ISR objects implicitly created (via **intConnect( )**) are named in that manner.

The *isrTag* is the means by which an ISR object is associated to an interrupt source. Typically this would be used to store the vector associated with the interrupt but this is not a requirement. The caller can use other identification schemes. This library does not make use of the *isrTag* other than for information providing/displaying purposes. ISR objects implicitly created due to a call to **intConnect( )** have their *isrTag* set to the value of the *vector* passed to **intConnect( )**.

The *handlerRtn* is the interrupt service routine that gets invoked when the associated interrupt occurs. The invocation includes passing *arg* to the *handlerRtn*. The prototype for the *handlerRtn* should be:

STATUS handlerRtn (int arg)

The *handlerRtn* and *arg* arguments for ISR objects that are implicitly created map directly to the *routine* and *parameter* arguments of the **intConnect( )** call.

The *options* argument is not use and must be set to zero by the caller.

The **CODING EXAMPLE** section found in the library description illustrates how to use this routine.

RETURNS

The **ISR_ID** of the ISR object created or **NULL** if creation failed

ERRNO

S_isrLib_ISR_NOT_INIT
ISR library must first be initialized

S_intLib_NOT_ISR_CALLABLE
this routine must not be called from an ISR

S_isrLib_INVALID_PARAM
options is not valid

SEE ALSO **isrLib**, **isrDelete( )**

isrDelete( )

NAME **isrDelete( )** – delete an ISR object

SYNOPSIS

```
STATUS isrDelete
    (
    ISR_ID isrId  /* ID of ISR object to delete */
    )
```

DESCRIPTION This routine destroys the ISR object specified by *isrId* and de-allocates the memory used by the object. This routine complements **isrCreate( )** and is meant to be used by code which disconnects routines from interrupt vectors by means other than calling **intDisconnect( )** since that routine implicitly deletes an ISR object.

WARNINGS Before deleting an ISR object, one must ensure it no longer plays a role in interrupt processing. That is, the ISR object must be disconnected from its interrupt source before being deleted.

An implicitly created ISR object, one created via **intConnect( )** that is, must never be explicitly deleted using **isrDelete( )**.

RETURNS **OK**, **ERROR** if the ISR object could not be deleted.

ERRNO

S_objLib_OBJ_ID_ERROR
isrId is not a valid ISR object

S_intLib_NOT_ISR_CALLABLE
this routine must not be called from an ISR

SEE ALSO **isrLib**, **isrCreate( )**

isrIdSelf( )

NAME **isrIdSelf( )** – get the ISR ID of the currently running ISR

SYNOPSIS

```
ISR_ID isrIdSelf (void)
```

DESCRIPTION This routine returns the ISR ID of the calling ISR. The ISR ID is **NULL** if the routine is called at task level. Calling this routine from a watchdog routine either returns the ISR ID of the system clock ISR or **NULL**. The latter is returned in cases where the processing of watchdog routines is deferred. This deferral only takes place when the system clock interrupts the kernel while it is in a critical section.

RETURNS The ISR ID of the calling ISR. Can be **NULL**.

ERRNO N/A

SEE ALSO **isrLib**

isrInfoGet( )

NAME **isrInfoGet( )** – get information about an ISR object

SYNOPSIS

```
STATUS isrInfoGet
    (
    ISR_ID     isrId,    /* ISR object ID                       */
    ISR_DESC * pIsrDesc  /* pointer to ISR description struct */
    )
```

DESCRIPTION This routine retrieves information regarding *idrId*. The information is returned in the **ISR_DESC** structure pointed to by *pIsrDesc*.

ISR_DESC is defined as:

```
typedef struct isr_desc
    {
    ISR_ID       isrId;             /* ISR_ID                             */
    char *       name;              /* name                               */
    UINT         isrTag;            /* interrupt tag                      */
    UINT         count;             /* # of times this ISR has been invoked */
    UINT         serviceCount;      /* # of times this ISR has returned OK  */
    UINT64       cpuTime;           /* cpu time spent in ISR              */
    int          options;           /* ISR object options                 */
    FUNCPTR      handlerRtn;        /* pointer to handler routine         */
    int          arg;               /* parameter to be passed to routine  */
    } ISR_DESC;
```

Note that because ISR time stamping functionality is not implemented, the *cpuTime* member of the structure is always 0.

RETURNS

OK, **ERROR** if *isrId* if not valid or *pIsrDesc* is **NULL**.

ERRNO

S_objLib_OBJ_ID_ERROR
isrId is not a valid ISR object

S_isrLib_INVALID_PARAM
pIsrDesc is not valid

SEE ALSO

isrLib

isrInvoke( )

NAME

isrInvoke( ) – invoke the handler routine of an ISR object

SYNOPSIS

```
STATUS isrInvoke
    (
    ISR_ID isrId
    )
```

DESCRIPTION

This routine invokes the handler routine of *isrId* as specified by the *handlerRtn* argument provided when the ISR object was created.

For implicitly created ISR objects, which are created as a result of a call to **intConnect( )**, **isrInvoke( )** is automatically called when the interrupt associated with the *vector* specified in **intConnect( )** occurs. Then, **isrInvoke( )** takes care of calling the user supplied handler routine for that interrupt. That is, the *routine* argument provided when **intConnect( )** was called.

For explicitly created ISR objects, this routine is meant to be installed as an interrupt handling routine by the creator of the ISR object such that when the associated interrupt occurs, it is automatically dispatched. The **CODING EXAMPLE** section of the library description illustrates how to use this routine in that manner.

Routine **isrInvoke( )** is therefore meant to be an intermediate routine between the hardware interrupt and the user provided handler for the interrupt. In essence, it is an instrumentation routine and this allows it to keep statistics on *isrId* such as the number of times its handler ran. A WINDVIEW events is also generated as a result of **isrInvoke( )** running.

RETURNS

OK, **ERROR** if *isrId* is not valid

ERRNO
S_objLib_OBJ_ID_ERROR
isrId is not a valid ISR object

SEE ALSO
isrLib, **isrCreate( )**

isrShow( )

NAME
isrShow( ) – show information about an ISR object

SYNOPSIS

```
STATUS isrShow
    (
    ISR_ID isrId  /* ID of ISR object or NULL for all */
    )
```

DESCRIPTION
This routine displays information related to an ISR object or all ISR objects.

EXAMPLE
A summary of a single ISR object is displayed as follows:

```
-> isrShow myIsrId
ID                    : 0x20f70568
Name                  : isrTestObject2
Interrupt Tag         : 1
Count                 : 0
Service Count         : 0
Options               : 0x0
Handler Routine       : 0x200c2750 (isrHandlerRtn)
Argument              : 0x0
```

A summary of all ISR objects is displayed as follows:

```
-> isrShow
ISR_ID     Name        Tag        Counts                HandlerRtn
---------- ----------- ---------- ---------------------
------------------------
0x20ffb178 isr1        4                     0/0        winIntRcv
0x20ffb100 isr2        0                    31/30       sysNvRamSe > +0x190
0x20f89610 isr3        3                     0/0        wdbPipePkt > +0x1ba4
0x20f705e8 isrTestO >  1                     0/0        isrHandlerRtn
0x20f70568 isrTestO >  1                     0/0        isrHandlerRtn
```

The format for the **Counts** column is count/serviceCount.

RETURNS
OK or **ERROR**

ERRNO
S_objLib_OBJ_ID_ERROR
isrId is not a valid ISR object

SEE ALSO
isrShow

kernelCpuEnable( )

NAME

kernelCpuEnable( ) – enable a CPU

SYNOPSIS

```
STATUS kernelCpuEnable
    (
    unsigned int cpuToEnable  /* CPU to enable */
    )
```

DESCRIPTION

This routine enables the CPU whose index matches the *cpuToEnable* argument. The value must be between 1 and N - 1, where N is the number of CPUs configured in the system. This figure can be obtained by calling **vxCpuConfiguredGet( )**. If successful, the call returns **OK**. Once a CPU is enabled, it starts dispatching tasks as per the scheduling algorithm. Furthermore, a subsequent call to **vxCpuEnabledGet( )** will include the specified CPU in the set of enabled CPUs.

By default VxWorks SMP enables all CPUs configured in the system at boot time. Therefore calling this routine is not required unless the default behaviour is overridden. This routine returns **ERROR** if the specified CPU is already enabled or is outside the range of configured CPUs. CPU 0 can never be enabled using this routine as it is the bootstrap CPU.

If this routine is unable to enable the specified CPU, before the timeout (**VX_ENABLE_CPU_TIMEOUT**) expires, the routine will return **ERROR**.

This routine always returns **ERROR** for VxWorks UP.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK or **ERROR** if cpu index is invalid or **VX_ENABLE_CPU_TIMEOUT** has been reached

ERRNO

N/A

SEE ALSO

kernelLib, **vxCpuConfiguredGet( )**, **vxCpuEnabledGet( )**

kernelInit( )

NAME

kernelInit( ) – initialize the kernel

SYNOPSIS

```
_WRS_FUNC_NORETURN void kernelInit
```

```
    (
    UINT32              sanity,   /* must match
_KERNEL_INIT_PARAMS_VN_AND_SIZE */
    _KERNEL_INIT_PARAMS *pParams  /* parameters */
    )
```

DESCRIPTION This routine initializes and starts the kernel. It should be called only once. The parameter *rootRtn* specifies the entry point of the user's start-up code that subsequently initializes system facilities (i.e., the I/O system, network). Typically, *rootRtn* is set to **usrRoot( )**.

Interrupts are enabled for the first time after **kernelInit( )** exits. VxWorks will not exceed the specified interrupt lock-out level during any of its brief uses of interrupt locking as a means of mutual exclusion.

The system memory partition is initialized by **kernelInit( )** with the size set by *pMemPoolStart* and *pMemPoolEnd*. Architectures that support a separate interrupt stack allocate a portion of memory for this purpose, of *intStackSize* bytes starting at *pMemPoolStart*.

NOTE SH77XX The interrupt stack is emulated by software, and it has to be located in a fixed physical address space (P1 or P2) if the on-chip MMU is enabled. If *pMemPoolStart* is in a logical address space (P0 or P3), the interrupt stack area is reserved on the same logical address space. The actual interrupt stack is relocated to a fixed physical space pointed by VBR.

RETURNS N/A

ERRNO N/A

SEE ALSO **kernelLib**, **intLockLevelSet( )**

kernelIsCpuIdle( )

NAME **kernelIsCpuIdle( )** – determine whether the specified CPU is idle

SYNOPSIS

```
BOOL kernelIsCpuIdle
    (
    unsigned int cpu  /* CPU to query status of */
    )
```

DESCRIPTION This routine returns **TRUE** if the specified CPU is idle.

For the uniprocessor VxWorks environment, this routine returns **TRUE** if the kernel is spinning in the idle loop, i.e. the kernel is not executing any tasks.

For SMP, this routine returns **TRUE** if the specified CPU is executing the idle task. If the specified CPU is not enabled, the CPU is considered to be idle and **TRUE** is returned. If the specified CPU is the one executing the calling task this routine returns **FALSE** since a CPU cannot be idle and executing a task other than the idle task. When called from ISR this routine returns **TRUE** if the interrupted task is the idle task. Otherwise **FALSE** is returned.

This routine is meant to be a debugging and system-monitoring tool.

RETURNS **TRUE** if the specified CPU is idle, **FALSE** otherwise

ERRNO N/A

SEE ALSO **kernelLib**, **kernelSystemIsIdle( )**

kernelIsSystemIdle( )

NAME **kernelIsSystemIdle( )** – determine whether all enabled processors are idle

SYNOPSIS

```
BOOL kernelIsSystemIdle (void)
```

DESCRIPTION For the uniprocessor VxWorks environment, this routine returns **TRUE** if the kernel is spinning in the idle loop, i.e. the kernel is not executing any tasks.

For SMP, this routine returns **TRUE** if all enabled processors are idle, or more specifically executing their respective idle task. CPUs that are not enabled are considered to be idle. Routine **vxCpuEnabledGet( )** can be used to determine the enabled CPUs in the system.

This routine is meant to be a debugging and system-monitoring tool.

RETURNS **TRUE** if all CPUs are idle, **FALSE** otherwise

ERRNO N/A

SEE ALSO **kernelLib**, **kernelIsCpuIdle( )**, **vxCpuEnabledGet( )**, **kernelCpuEnable( )**

kernelRoundRobinInstall( )

NAME **kernelRoundRobinInstall( )** – install VxWorks Round Robin implementation

SYNOPSIS

```
STATUS kernelRoundRobinInstall(void)
```

DESCRIPTION

This routine allows user custom schedulers to take advantage of the vxWorks implementation of the round robin scheduling policy. This routine should only be used if the component **INCLUDE_CUSTOM_SCHEDULER** is configured and the user wants to take advantage of the VxWorks round robin policy. Below is an example of its usage:

```
  usrCustomSchedulerInit (void)
      {
     ...
     tickAnnounceHook (usrTickFunc); /* register custom hook func */
     kernelRoundRobinInstall();      /* install the VxWorks round robin */
      ...
      }

   usrTickFunc (int tid)
      {
      ...
      if (_func_kernelRoundRobinHook)
          _func_kernelRoundRobinHook (tid);
      ...
      }
```

RETURNS

OK, or **ERROR** if _func_kernelRoundRobinHook has been initialized

ERRNO

N/A

SEE ALSO

kernelLib, **usrCustomScheduler.c**

kernelTimeSlice( )

NAME

kernelTimeSlice( ) – enable round-robin selection

SYNOPSIS

```
STATUS kernelTimeSlice
    (
    int ticks  /* time-slice in ticks or 0 to disable round-robin */
    )
```

DESCRIPTION

This routine enables round-robin selection among tasks of same priority and sets the system time-slice to *ticks*. Round-robin scheduling is disabled by default. A time-slice of zero ticks disables round-robin scheduling.

A hook routine, **kernelRoundRobinHook( )**, is installed by this routine. **kernelRoundRobinHook( )** is the routine that performs the bulk of the work to schedule tasks in a round-robin fashion. This hook is called at each tick interrupt when **kernelTimeSlice( )** is called for the first time if the system is configured with **INCLUDE_VX_NATIVE_SCHEDULER**.

For more information about round-robin scheduling, see the manual entry for **kernelLib**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if **kernelRoundRobinHook( )** can not be installed

ERRNO

N/A

SEE ALSO

kernelLib

kernelVersion( )

NAME

kernelVersion( ) – return the WIND kernel revision string

SYNOPSIS

```
char *kernelVersion (void)
```

DESCRIPTION

This routine returns a string which contains the current revision of the WIND kernel. The string is of the form "WIND version x.y", where "x" corresponds to the kernel major revision, and "y" corresponds to the kernel minor revision.

RETURNS

A pointer to a string of format "WIND version x.y"

ERRNO

N/A

SEE ALSO

kernelLib

kill( )

NAME

kill( ) – send a signal to a task (POSIX)

SYNOPSIS

```
int kill
    (
    int tid,   /* task to send signal to */
    int signo  /* signal to send to task */
    )
```

DESCRIPTION

This routine sends a signal *signo* to the task specified by *tid*.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **taskKill( )**

l( )

NAME **l( )** – disassemble and display a specified number of instructions

SYNOPSIS

```
void l
    (
    INSTR * addr,  /* address of first instruction to disassemble */
                   /* if 0, continue from the last instruction */
                   /* disassembled on the last call to l */
    int     count  /* number of instruction to disassemble */
                   /* if 0, use the same as the last call to l */
    )
```

DESCRIPTION This routine disassembles a specified number of instructions and displays them on standard output. If the address of an instruction is entered in the system symbol table, the symbol will be displayed as a label for that instruction. Also, addresses will be displayed symbolically.

To execute, enter:

```
    -> l [address [,count]]
```

If *address* is omitted or zero, disassembly continues from the previous address. If *count* is omitted or zero, the last specified count is used (initially 10). As with all values entered via the shell, the address may be typed symbolically.

RETURNS N/A

ERRNO N/A

SEE ALSO **dbgLib**, **d( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

l0( )

NAME **l0( )** – return the contents of register l0 (also l1-l7) (SimSolaris)

SYNOPSIS

```
int l0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of local register l0 from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all local registers (l0 - l7): **l0( )** - **l7( )**.

RETURNS The contents of register l0 (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

ld( )

NAME **ld( )** – load an object module into memory

SYNOPSIS

```
MODULE_ID ld
    (
    int    syms,      /* -1, 0, or 1 */
    BOOL   noAbort,   /* TRUE = don't abort script on error */
    char * name       /* name of object module, NULL = standard input */
    )
```

DESCRIPTION This command loads an object module from a file or from standard input. The object module must be in architecture object module format (OMF). For most of the architectures, this is ELF format. External references in the module are resolved during loading. The *syms* parameter determines how symbols are loaded; possible values are:

0
: Add global symbols to the system symbol table.

1
: Add global and local symbols to the system symbol table.

-1
: Add no symbols to the system symbol table.

If there is an error during loading (e.g., externals undefined, too many symbols, etc.), then **shellScriptAbort( )** is called to stop any script that this routine was called from. If *noAbort* is **TRUE**, errors are noted but ignored.

The normal way of using **ld( )** is to load all symbols (*syms* = 1) during debugging and to load only global symbols later.

NOTE

The routine **ld( )** is a **shell routine**. That is, it is designed to be used only from the shell, and not in code running on the target. In future releases, calling **ld( )** directly from code may not be supported.

COMMON SYMBOLS

On the kernel shell, for the **ld( )** routine only, common symbol behavior is determined by the value of the global variable **ldCommonMatchAll**. The reasoning for **ldCommonMatchAll** matches the purpose of the windsh environment variable, **LD_COMMON_MATCH_ALL** as explained below.

If **ldCommonMatchAll** is set to **TRUE** (equivalent to windsh "**LD_COMMON_MATCH_ALL**=on"), the loader tries to match a common symbol with an existing one. If a symbol with the same name is already defined, the loader takes its address. Otherwise, the loader creates a new entry. If set to **FALSE** (equivalent to windsh "**LD_COMMON_MATCH_ALL**=off"), the loader does not try to find an existing symbol. It creates an entry for each common symbol.

EXAMPLE

The following example loads the ELF file "module" from the default file device into memory, and adds any global symbols to the symbol table:

```
    -> ld < module
```

This example loads "**test.o**" with all symbols:

```
    -> ld 1,0,"test.o"
```

RETURNS

a **MODULE_ID**, or **NULL** if there are too many symbols, the object file format is invalid, or there is an error reading the file.

ERRNO

open( ) errnos, **loadModule( )** errnos.

SEE ALSO

usrLib, **loadLib**, **unld( )**, **reld( )**, the VxWorks programmer guides.

ledClose( )

NAME

ledClose( ) – discard the line-editor ID

SYNOPSIS

```
STATUS ledClose
```

```
    (
    FAST LED_ID ledId  /* ID returned by ledOpen */
    )
```

DESCRIPTION This routine frees resources allocated by **ledOpen( )**. The low-level input/output file descriptors are not closed.

RETURNS **OK**, or **ERROR** if *ledId* is invalid.

ERRNO N/A

SEE ALSO **ledLib**, **ledOpen( )**

ledControl( )

NAME **ledControl( )** – change the line-editor ID parameters

SYNOPSIS

```
void ledControl
    (
    FAST LED_ID ledId,    /* ID returned by ledOpen */
    int         inFd,     /* new input fd (NONE = no change) */
    int         outFd,    /* new output fd (NONE = no change) */
    int         histSize  /* new hist list size (NONE=no change),
(0=display)*/
    )
```

DESCRIPTION This routine changes the input/output file descriptor and the size of the history list.

RETURNS N/A

ERRNO N/A

SEE ALSO **ledLib**

ledLibInit( )

NAME **ledLibInit( )** – initialize the line editing facilities

SYNOPSIS

```
STATUS ledLibInit (void)
```

DESCRIPTION This routine initializes the line editing facilities. It is called once from **shellLibInit( )**.

RETURNS **OK** or **ERROR** if there was a problem

ERRNO Not Available

SEE ALSO **ledLib**

ledOpen( )

NAME **ledOpen( )** – create a new line-editor ID

SYNOPSIS

```
LED_ID ledOpen
    (
    int inFd,      /* low-level device input fd */
    int outFd,     /* low-level device output fd */
    int histSize  /* size of history list */
    )
```

DESCRIPTION This routine creates the ID that is used by **ledRead( )**, **ledClose( )**, and **ledControl( )**. Storage is allocated for up to *histSize* previously read lines.

RETURNS The line-editor ID, or **ERROR** if the routine runs out of memory.

ERRNO N/A

SEE ALSO **ledLib**, **ledRead( )**, **ledClose( )**, **ledControl( )**.

ledRead( )

NAME **ledRead( )** – read a line with line-editing

SYNOPSIS

```
int ledRead
    (
    LED_ID ledId,     /* ID returned by ledOpen */
    char * string,    /* where to return line */
    UINT   maxBytes   /* maximum number of chars to read */
    )
```

DESCRIPTION This routine handles line-editing and history substitutions. If the low-level input file descriptor is not in **OPT_LINE** mode, only an ordinary **read( )** routine will be performed.

RETURNS the number of characters read, or **EOF**.

ERRNO N/A

SEE ALSO **ledLib**, **ledOpen( )**

link( )

NAME **link( )** – link a file

SYNOPSIS

```
int link
    (
    const char *name,    /* name of file to be linked          */
    const char *newname  /* name with which to link */
    )
```

DESCRIPTION This routine links the name of a file from *newname* to *name*.

RETURNS **OK**, or **ERROR** if the file could not be opened or linked.

ERRNO

ENOENT
Either name or newname is an empty string.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in path.

others
Other errors reported by device driver.

SEE ALSO **fsPxLib**

lio_listio( )

NAME **lio_listio( )** – initiate a list of asynchronous I/O requests (POSIX)

SYNOPSIS

```
int lio_listio
    (
    int                  mode,    /* LIO_WAIT or LIO_NOWAIT */
    struct aiocb *const list[],   /* list of operations */
    int                  nEnt,    /* size of list */
    struct sigevent *   pSig     /* signal on completion */
    )
```

DESCRIPTION

This routine submits a number of I/O operations (up to **AIO_LISTIO_MAX**) to be performed asynchronously. *list* is a pointer to an array of **aiocb** structures that specify the AIO operations to be performed. The array is of size *nEnt*.

The **aio_lio_opcode** field of the **aiocb** structure specifies the AIO operation to be performed. Valid entries include **LIO_READ**, **LIO_WRITE**, and **LIO_NOP**. **LIO_READ** corresponds to a call to **aio_read( )**, **LIO_WRITE** corresponds to a call to **aio_write( )**, and **LIO_NOP** is ignored.

The *mode* argument can be either **LIO_WAIT** or **LIO_NOWAIT**. If *mode* is **LIO_WAIT**, **lio_listio( )** does not return until all the AIO operations complete and the *pSig* argument is ignored. If *mode* is **LIO_NOWAIT**, the **lio_listio( )** returns as soon as the operations are queued. In this case, if *pSig* is not **NULL** and the signal number indicated by **pSig->sigev_signo** is not zero, the signal **pSig->sigev_signo** is delivered when all requests have completed.

RETURNS

OK if requests queued successfully, otherwise **ERROR**.

ERRNO

EINVAL
EAGAIN
EIO

SEE ALSO

aioPxLib, **aio_read( )**, **aio_write( )**, **aio_error( )**, **aio_return( )**.

lkAddr( )

NAME

lkAddr( ) – list symbols whose values are near a specified value

SYNOPSIS

```
void lkAddr
    (
    unsigned int addr  /* address around which to look */
    )
```

DESCRIPTION

This command lists the symbols in the system symbol table that are near a specified value. The symbols that are displayed include:

- symbols whose values are immediately less than the specified value
- symbols with the specified value
- succeeding symbols, until at least 12 symbols have been displayed

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by **ld( )**.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **symLib**, **symEach( )**, the VxWorks programmer guides.

lkup( )

NAME **lkup( )** – list symbols

SYNOPSIS

```
void lkup
    (
    char *substr  /* substring to match */
    )
```

DESCRIPTION This command lists all symbols in the system symbol table whose names contain the string *substr*. If *substr* is omitted or is 0, a short summary of symbol table statistics is printed. If *substr* is the empty string (""), all symbols in the table are listed.

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by **ld( )**.

By default, **lkup( )** displays 22 symbols at a time. This can be changed by modifying the global variable **symLkupPgSz**. If this variable is set to 0, **lkup( )** displays all the symbols without interruption.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **symLib**, **symEach( )**, the VxWorks programmer guides.

ll( )

NAME **ll( )** – generate a long listing of directory contents

SYNOPSIS

```
STATUS ll
    (
    const char * dirName  /* name of directory to list */
    )
```

DESCRIPTION This command causes a long listing of a directory's contents to be displayed. It is equivalent to:

```
    -> dirList 1, dirName, 1, 0
```

dirName is a name of a directory or file, and may contain wildcards.

NOTE 1 This is a target resident function, which manipulates the target I/O system. It must be preceded with the @ letter if executed from the Host Shell (windsh), which has a built-in command of the same name that operates on the Host's I/O system.

NOTE 2 When used with **netDrv** devices (FTP or RSH), **ll( )** does not give directory information. It is equivalent to an **ls( )** call with no long-listing option.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dirList( )**, the VxWorks programmer guides.

llr( )

NAME **llr( )** – do a long listing of directory and all its subdirectories contents

SYNOPSIS

```
STATUS llr
    (
    const char * dirName  /* name of directory to list */
    )
```

DESCRIPTION This command causes a long listing of a directory's contents to be displayed. It is equivalent to:

```
    -> dirList 1, dirName, 1, 0
```

dirName is a name of a directory or file, and may contain wildcards.

NOTE When used with **netDrv** devices (FTP or RSH), **ll( )** does not give directory information. It is equivalent to an **ls( )** call with no long-listing option.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dirList( )**, the VxWorks programmer guides.

lnPciRegister( )

NAME
lnPciRegister( ) – register with the VxBus subsystem

SYNOPSIS
```
void lnPciRegister(void)
```

DESCRIPTION
This routine registers the PCnet/PCI driver with VxBus as a child of the PCI bus type.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
am79c97xVxbEnd

loadModule( )

NAME
loadModule( ) – load an object module into memory

SYNOPSIS
```
MODULE_ID loadModule
    (
    int fd,      /* file descriptor of file to load */
    int options  /* symbols to add to table, other behavior options */
    )
```

DESCRIPTION
This routine loads an object module from the specified file, and places the TEXT, DATA, and BSS into memory allocated from the system memory pool (i.e. the heap).

Calling this function is equivalent to calling **loadModuleAt( )** with **NULL** for the addresses of TEXT, DATA, and BSS segments. For more details as well as the supported option flags, see the reference entry for **loadModuleAt( )**.

RETURNS
The **MODULE_ID**, or **NULL** if there was a problem. **NULL** is also returned if there are undefined symbols, but the module is not unloaded. (See **loadModuleAt( )** for more details).

ERRNO
Not Available

SEE ALSO
loadLib, **loadModuleAt( )**

loadModuleAt( )

NAME

loadModuleAt( ) – load an object module into memory

SYNOPSIS

```
MODULE_ID loadModuleAt
    (
    int     fd,       /* file descriptor from which to read module */
    int     options,  /* symbols to add to table, other behavior options */
    char ** ppText,   /* load TEXT segment at addr. pointed to by this */
                      /* pointer, return load addr. via this ptr */
    char ** ppData,   /* load DATA segment at addr. pointed to by this */
                      /* pointer, return load addr. via this ptr */
    char ** ppBss     /* load BSS segment at addr. pointed to by this */
                      /* pointer, return load addr. via this ptr */
    )
```

DESCRIPTION

This routine reads an object module from a file descriptor (the *fd* parameter) and loads the TEXT, DATA and BSS segments into the system memory space. The code is properly relocated according to the relocation commands found in the ELF file.

Unresolved references to external symbols will be linked to symbols found in the system. Symbols in the object module being loaded can optionally be added to a symbol table.

It is also possible to give a specific load directive for each loadable segment, in which case the loader can use memory that the user set aside using **malloc( )** or **memalign( )**.

The exact loader behavior can be controlled using load flags, as described below. Load flags may be combined (binary OR) to the extent that they are not mutually exclusive.

LINKING UNRESOLVED EXTERNALS

As the module is loaded, any unresolved external references are resolved by looking up the symbols in the system symbol table. If found, the references to those functions or data in the new module are linked to the symbols found in the system symbol table. If there is more than one possible match in the system symbol table, the symbol encountered first (which is the one added most recently) will be used.

If an unresolved external reference cannot be found in the system symbol table, then an error message ("undefined symbol: ...") is printed for the symbol, but the loading and linking continues. The partially resolved module is not removed, to enable the user to examine the module for debugging purposes. Care should be taken when executing code from the module. Executing code which contains references to unresolved symbols may have unexpected results and may corrupt the system memory.

Even though a module with unresolved symbols remains loaded after this routine returns, **NULL** will be returned to enable the caller to detect the failure programatically. To unload the module, the caller may either call the unload routine with the module name, or look up the module using the module name and then unload the module using the returned **MODULE_ID**. See the library entries for **moduleLib** and **unldLib** for more details.

FULLY LINKED MODULES

The VxWorks kernel loader supports loading of fully linked modules via the **LOAD_FULLY_LINKED** flag. Fully linked modules do not contain any relocations and are statically linked on the host to run at a fixed address.

As the loader has no facility to allocate memory at a particular address, it is up to the user to configure his/her board so that the addresses are available for the loader to copy the module to. The information contained in the ELF headers can be used for that purpose. It can be printed by using the "-l" option flag of the GNU "readelf" tool (segment addresses, sizes and alignement requirements). Please also refer to the alignement and memory protection sections below for important information on how to allocate memory for use by the loader.

Since the host linker will store the section/segment addresses and sizes at link time in the object file, there is no need to provide them to the loader. It will pick them up automatically from the ELF headers.

Symbol tables are supported (see the symbol table dedicated section below) but not mandatory.

ADDING SYMBOLS TO THE SYMBOL TABLE

The symbols defined in the module to be loaded may be added to the system symbol table; this behaviour is controlled by the value of the *options* parameter:

LOAD_NO_SYMBOLS
Add no symbols to the system symbol table.

LOAD_LOCAL_SYMBOLS
Add only local symbols to the system symbol table.

LOAD_GLOBAL_SYMBOLS
Add only external symbols to the system symbol table.

LOAD_ALL_SYMBOLS
Add both local and external symbols to the system symbol table.

When the *options* parameter is left unspecified (**NULL**), the loader defaults to **LOAD_GLOBAL_SYMBOLS**. If the module symbols are added to the system symbol table, modules loaded later may link against these symbols. There is no way to make a module symbols available for debugging and, at the same time, prevent other modules from linking against those symbols.

CODE MODULE VISIBILITY

By default any object module loaded in the system will appear as a code module and be visible with commands such as **moduleShow( )**. It is however possible to hide a code module with the flag:

HIDDEN_MODULE
Do not display the module via **moduleShow( )**.

RELOCATION

The relocation commands in the object module are used to relocate the TEXT, DATA, and BSS segments of the module. The ELF file sections are sorted into these three types of segments according to the ELF section flags. The location of each segment can either be specified explicitly or left unspecified, in which case memory will be allocated for the segment from the kernel heap, according to the section category.

To specify where one or more of the segments should be installed, use the parameters *ppText*, *ppData*, and *ppBss*. Each of these can have either of the following values:

NULL
No load address is specified, none will be returned.

A pointer to **LD_NO_ADDRESS**
No load address is specified; after the load is performed, **LD_NO_ADDRESS** will be replaced by the actual segment load address.

A pointer to an address
The load address is specified.

The *ppText*, *ppData*, and *ppBss* parameters specify where to load the TEXT, DATA, and BSS segments, respectively. Each of these parameters is a pointer to a pointer; for example, ***ppText* gives the address where the TEXT segment is to begin.

Note that it is up to the user to reserve a sufficient amount of memory for each segment which address is specified. In particular, alignement requirements need to be kept in mind when reserving memory for a segment (see the alignement-dedicated section below for more information). Finally, remember that the loader will only free memory it allocates. This means the user will have to free memory he/she reserved when, for instance, a module is unloaded.

For any of the three parameters, there are two ways to request that new memory be allocated, rather than specifying the segment starting address: you can either specify the parameter itself as **NULL**, or you can write the constant **LD_NO_ADDRESS** in place of an address. In the second case, the **loadModuleAt( )** routine replaces the **LD_NO_ADDRESS** value with the address actually used for each section (that is, it records the address at **ppText*, **ppData*, or **ppBss*).

The double indirection not only permits reporting the addresses actually used, but also allows you to specify loading a segment at the beginning of memory, since the following cases can be distinguished:

- Allocate memory for a segment (TEXT in this example): *ppText* == **NULL**
- Begin a section at address zero (the TEXT section, below): **ppText* == 0

Note that calling **loadModule( )** is equivalent to calling **loadModuleAt( )** with all three segment-address parameters set to **NULL**.

COMMON

Some host compiler/linker combinations use another storage class internally called *COMMON*. In the C language, uninitialized global variables are eventually put in the BSS segment. However, in partially linked object modules they are flagged internally as

COMMON and the static linker (host) resolves these and places them in BSS as a final step in creating a fully linked object module. However, the kernel loader is used to load partially linked object modules into the kernel, not executable modules. When the VxWorks loader encounters a variable labeled as COMMON, memory for the variable can be allocated (see below), and the variable is entered in the symbol table (if specified) at that address.

Note that most UNIX loaders have an option that forces resolution of the COMMON storage while leaving the module relocatable. For example, with typical BSD UNIX loaders, "-d" serves that purpose (in conjunction with "-r" to generate relocatable output). The GNU linker, ld, belongs to this category. With DIAB, option "-a" has to be passed to the dld linker (again, in conjunction with "-r" to generate relocatable output).

When the kernel loader encounters a variable labeled "COMMON", its behavior depends on the following flags:

LOAD_COMMON_MATCH_NONE
Allocate memory for the variable with **malloc( )** and enter the variable in the target symbol table (if specified) with type **SYM_COMM** at that address. This is the default. Note that the remark about symbol linking visibility also applies here : unless **LOAD_NO_SYMBOLS** is set, **LOAD_COMMON_MATCH_NONE** won't prevent other modules from linking against the module symbols.

LOAD_COMMON_MATCH_USER
The loader seeks a matching "user" symbol, i.e. symbols that have been added by dynamically loaded object modules (*vs* symbols statically present at boot time). If no matching symbol exists, it acts like **LOAD_COMMON_MATCH_NONE**. If several matching symbols exist, the symbol most recently added to the target symbol table is used.

LOAD_COMMON_MATCH_ALL
The loader seeks for any matching symbol. All symbols are considered. If no matching symbol exists, then it acts like **LOAD_COMMON_MATCH_NONE**. If several matches are found, the preference order is the same as for **LOAD_COMMON_MATCH_USER**.

C++ CONSTRUCTORS SUPPORT

The loader applies the C++ strategy as defined by the C++ runtime library at the time of the load operation. If this strategy is set to "automatic", then the C++ constructors are executed. If this strategy is set to "manual", then the loader does not execute the C++ constructors (in that case, they can be called manually via the **cplusCallCtors( )** API). It is possible to prevent this default behavior with the following flags:

LOAD_CPLUS_XTOR_AUTO
When this flag is set, the loader always executes the C++ constructors associated to the module.

LOAD_CPLUS_XTOR_MANUAL
When this flag is set, the loader never executes the C++ constructors associated to the module.

Note that if there are undefined symbols in the module, the loader will not run the C++ constructors regardless of the status of the above flags (this is to prevent erratic behavior/application crashes).

Please refer to **unldLib** for the corresponding unloader flags.

WEAK SYMBOL HANDLING

Most ELF symbols use the standard global/local symbol binding. Some languages, however, make use of another binding called WEAK. When a WEAK symbol is encountered, the loader behavior depends on the following flags:

LOAD_WEAK_MATCH_ALL
The loader looks for an already existing global definition with the same name in the symbol table. If one can be found, the loader honors the existing definition and ignores the WEAK one. If no match can be found, the loader behaves like **LOAD_WEAK_MATCH_NONE**. This is the default.

LOAD_WEAK_MATCH_NONE
The WEAK symbol is registered in the symbol table (if specified) as a global symbol regardless of any existing definition. This behavior matches what was done by the VxWorks 5.x loader.

ALIGNMENT CONSIDERATIONS

Please note that memory required to load a segment is more than the bare sum of the size of sections it contains. The memory needed to load a segment is the sum of the sizes of its sections plus the padding required by each section alignment requirements (if a section has no alignment requirements, the architecture default alignment is taken as the section alignment requirement). Sections alignment requirements can be visualized using the GNU "objdump" tool (part of the binutils) with the "-h" option (alignment requirements are represented as powers of 2).

MEMORY PROTECTION

When memory protection is available (see **vmBaseLib** for more information), the loader will write-protect the module TEXT segment. For this to properly work, the TEXT segment size will be rounded up to the next page of memory and the TEXT segment address will be requested aligned on a page boundary. The unloader will also accordingly unprotect the TEXT segment when unloading the module.

Please note that the loader will only protect the TEXT segment (and the unloader unprotect it) if it itself allocates memory for it. If **loadModuleAt( )** is used with user-reserved memory instead (or when loading a fully linked module), it is up to the user to properly write-protect/unprotect the module TEXT segment. Here is how to do it :

Allocate aligned TEXT segment memory rounded up to the next page :

```
protectedTextSize = ROUND_UP (textSize, VM_PAGE_SIZE_GET());
pText = (char *) memalign (VM_PAGE_SIZE_GET(), protectedTextSize);
```

Write-protect the TEXT segment after it has been loaded by **loadModuleAt( )** :

```
    VM_STATE_SET (NULL, pText, protectedTextSize, VM_STATE_MASK_WRITABLE,
                  VM_STATE_WRITABLE_NOT);
```

Later, you may want to unprotect it :

```
    VM_STATE_SET (NULL, pText, protectedTextSize, VM_STATE_MASK_WRITABLE,
                  VM_STATE_WRITABLE);
```

EXAMPLES

These examples are of ways to invoke the loader from C code. To use the loader from the shell, use the loader shell commands, defined in **usrLib**.

Load a module into memory allocated by the loader:

```
    module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, NULL, NULL, NULL);
```

Load a module into memory allocated by the loader and retrieve segment addresses:

```
    pText = pData = pBss = LD_NO_ADDRESS;
    module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, &pText, &pData,
&pBss);
```

Load a module to off-board memory at a specified address:

```
    pText = 0x800000;                   /* address of TEXT segment
*/
    pData = pBss = LD_NO_ADDRESS;       /* other segments allocated by loader
*/
    module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, &pText, &pData,
&pBss);
```

RETURNS

A **MODULE_ID**, or **NULL** if there was a problem.

ERRNO

Possible errnos set by this routine include:

+ **S_loadLib_INVALID_ARGUMENT**

For a complete description of the errnos, see the reference documentation for **loadLib**.

SEE ALSO

loadLib, **unldLib**

log10f( )

NAME

log10f( ) – compute a base-10 logarithm (ANSI)

SYNOPSIS

```
float log10f
    (
    float x  /* value to compute the base-10 logarithm of */
    )
```

DESCRIPTION

This routine returns the base-10 logarithm of *x* in single precision.

RETURNS The single-precision base-10 logarithm of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

log2( )

NAME **log2( )** – compute a base-2 logarithm

SYNOPSIS

```
double log2
    (
    double x  /* value to compute the base-two logarithm of */
    )
```

DESCRIPTION This routine returns the base-2 logarithm of *x* in double precision.

RETURNS The double-precision base-2 logarithm of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

log2f( )

NAME **log2f( )** – compute a base-2 logarithm

SYNOPSIS

```
float log2f
    (
    float x  /* value to compute the base-2 logarithm of */
    )
```

DESCRIPTION This routine returns the base-2 logarithm of *x* in single precision.

RETURNS The single-precision base-2 logarithm of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

logFdAdd( )

NAME

logFdAdd( ) – add a logging file descriptor

SYNOPSIS

```
STATUS logFdAdd
    (
    int fd  /* file descriptor for additional logging device */
    )
```

DESCRIPTION

This routine adds to the log file descriptor list another file descriptor *fd* to which messages will be logged. The file descriptor must be a valid open file descriptor.

RETURNS

OK, or **ERROR** if the allowable number of additional logging file descriptors (5) is exceeded.

ERRNO

Not Available

SEE ALSO

logLib, **logFdDelete( )**

logFdDelete( )

NAME

logFdDelete( ) – delete a logging file descriptor

SYNOPSIS

```
STATUS logFdDelete
    (
    int fd  /* file descriptor to stop using as logging device */
    )
```

DESCRIPTION

This routine removes from the log file descriptor list a logging file descriptor added by **logFdAdd( )**. The file descriptor is not closed; but is no longer used by the logging facilities.

RETURNS

OK, or **ERROR** if the file descriptor was not added with **logFdAdd( )**.

ERRNO

Not Available

SEE ALSO

logLib, **logFdAdd( )**

logFdSet( )

NAME

logFdSet( ) – set the primary logging file descriptor

SYNOPSIS

```
void logFdSet
    (
    int fd  /* file descriptor to use as logging device */
    )
```

DESCRIPTION

This routine changes the file descriptor where messages from **logMsg( )** are written, allowing the log device to be changed from the default specified by **logInit( )**. It first removes the old file descriptor (if one had been previously set) from the log file descriptor list, then adds the new *fd*.

The old logging file descriptor is not closed or affected by this call; it is simply no longer used by the logging facilities.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

logLib, **logFdAdd( )**, **logFdDelete( )**

logInit( )

NAME

logInit( ) – initialize message logging library

SYNOPSIS

```
STATUS logInit
    (
    int fd,      /* file descriptor to use as logging device */
    int maxMsgs  /* max. number of messages allowed in log queue */
    )
```

DESCRIPTION

This routine specifies the file descriptor to be used as the logging device and the number of messages that can be in the logging queue. If more than *maxMsgs* are in the queue, they will be discarded. A message is printed to indicate lost messages.

This routine spawns **logTask( )**, the task-level portion of error logging.

This routine must be called before any other routine in **logLib**. This is done by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS

OK, or **ERROR** if a message queue could not be created or **logTask( )** could not be spawned.

ERRNO Not Available

SEE ALSO **logLib**

logMsg( )

NAME **logMsg( )** – log a formatted error message

SYNOPSIS

```
int logMsg
    (
    char *fmt,  /* format string for print */
    int  arg1,  /* first of six required args for fmt */
    int  arg2,
    int  arg3,
    int  arg4,
    int  arg5,
    int  arg6
    )
```

DESCRIPTION This routine logs a specified message via the logging task. This routine's syntax is similar to **printf( )** -- a format string is followed by arguments to format. However, the **logMsg( )** routine takes a char * rather than a const char * and requires a fixed number of arguments (6).

The task ID of the caller is prepended to the specified message.

SPECIAL CONSIDERATIONS

Because **logMsg( )** does not actually perform the output directly to the logging streams, but instead queues the message to the logging task, **logMsg( )** can be called from interrupt service routines.

However, since the arguments are interpreted by the **logTask( )** at the time of actual logging, instead of at the moment when **logMsg( )** is called, arguments to **logMsg( )** should not be pointers to volatile entities (e.g., dynamic strings on the caller stack).

logMsg( ) checks to see whether or not it is running in interupt context. If it is, it will not block. However, if invoked from a task, it can cause the task to block.

For more detailed information about the use of **logMsg( )**, see the manual entry for **logLib**.

EXAMPLE If the following code were executed by task 20:

```
{
name = "GRONK";
num = 123;

logMsg ("ERROR - name = %s, num = %d.\\n", name, num, 0, 0, 0, 0);
```

```
    }
```

the following error message would appear on the system log:

```
    0x180400 (t20): ERROR - name = GRONK, num = 123.
```

RETURNS The number of bytes written to the log queue, or **EOF** if the routine is unable to write a message.

ERRNO Not Available

SEE ALSO **logLib**, **printf( )**, **logTask( )**

logTask( )

NAME **logTask( )** – message-logging support task

SYNOPSIS

```
void logTask (void)
```

DESCRIPTION This routine prints the messages logged with **logMsg( )**. It waits on a message queue and prints the messages as they arrive on the file descriptor specified by **logInit( )** (or a subsequent call to **logFdSet( )** or **logFdAdd( )**).

This task is spawned by **logInit( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **logLib**, **logMsg( )**

logf( )

NAME **logf( )** – compute a natural logarithm (ANSI)

SYNOPSIS

```
float logf
    (
    float x  /* value to compute the natural logarithm of */
    )
```

DESCRIPTION This routine returns the logarithm of *x* in single precision.

RETURNS The single-precision natural logarithm of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

loginDefaultEncrypt( )

NAME **loginDefaultEncrypt( )** – default password encryption routine

SYNOPSIS

```
STATUS loginDefaultEncrypt
    (
    char * in,  /* input string */
    char * out  /* encrypted string */
    )
```

DESCRIPTION This routine provides default encryption for login passwords. It employs a simple encryption algorithm. It takes as arguments a string *in* and a pointer to a buffer *out*. The encrypted string is then stored in the buffer.

The input strings must be at least 8 characters and no more than 40 characters.

If a more sophisticated encryption algorithm is needed, this routine can be replaced, as long as the new encryption routine retains the same declarations as the default routine. The routine **vxencrypt** in **host/***hostOs*/bin should also be replaced by a host version of *encryptionRoutine*. For more information, see the manual entry for **loginEncryptInstall( )**.

RETURNS **OK**, or **ERROR** if the password is invalid.

ERRNO Possible errnos set by this routine include:

S_loginLib_INVALID_PASSWORD
in string is not a valid password string.

SEE ALSO **loginLib**, **loginEncryptInstall( )**, **vxencrypt**

loginEncryptInstall( )

NAME **loginEncryptInstall( )** – install an encryption routine

SYNOPSIS

```
void loginEncryptInstall
```

```
    (
    FUNCPTR rtn,  /* function pointer to encryption routine */
    int     var   /* argument to the encryption routine (unused) */
    )
```

DESCRIPTION This routine allows the user to install a custom encryption routine. The custom routine *rtn* must be of the following form:

```
STATUS encryptRoutine
       (
       char *password,               /* string to encrypt    */
       char *encryptedPassword       /* resulting encryption */
       )
```

The encryption result string must remain less or equal to **MAX_PASSWORD_LEN**.

When a custom encryption routine is installed, a host version of this routine must be written to replace the tool **vxencrypt** in **host/***hostOs*/bin.

EXAMPLE The custom example above could be installed as follows:

```
#ifdef INCLUDE_SECURITY
    loginInit ();                               /* initialize login table   */
    shellLoginInstall (loginPrompt, NULL);      /* install shell security   */
    loginEncryptInstall (encryptRoutine, NULL);
                                                /* install encrypt. routine */
#endif
```

RETURNS N/A

ERRNO N/A

SEE ALSO **loginLib**, **loginDefaultEncrypt( )**, **vxencrypt**

loginInit( )

NAME **loginInit( )** – initialize the login table

SYNOPSIS

```
void loginInit (void)
```

DESCRIPTION This routine must be called to initialize the login data structure used by routines throughout this module. If the configuration macro **INCLUDE_SECURITY** is defined, it is called by **usrRoot( )** in **usrConfig.c**, before any other routines in this module.

RETURNS N/A

ERRNO N/A

SEE ALSO **loginLib**

loginPrompt( )

NAME **loginPrompt( )** – display a login prompt and validate a user entry

SYNOPSIS

```
STATUS loginPrompt
    (
    char * userName  /* user name, ask if NULL or not provided */
    )
```

DESCRIPTION This routine displays a login prompt and validates a user entry. If both user name and password match with an entry in the login table, the user is then given access to the VxWorks system. Otherwise, it prompts the user again.

All control characters are disabled during authentication except CTRL-D, which will terminate the remote login session.

RETURNS **OK** if the name and password are valid, or **ERROR** if there is an **EOF** or the routine times out.

ERRNO N/A

SEE ALSO **loginLib**

loginStringSet( )

NAME **loginStringSet( )** – change the login string

SYNOPSIS

```
void loginStringSet
    (
    char * newString  /* string to become new login prompt */
    )
```

DESCRIPTION This routine changes the login prompt string to *newString*. The maximum string length is **MAX_LOGIN_NAME_LEN** characters.

RETURNS N/A

ERRNO N/A

SEE ALSO **loginLib**

loginUserAdd( )

NAME **loginUserAdd( )** – add a user to the login table

SYNOPSIS

```
STATUS loginUserAdd
    (
    char name[MAX_LOGIN_NAME_LEN + 1],  /* user name */
    char passwd[MAX_PASSWORD_LEN + 1]   /* user password */
    )
```

DESCRIPTION This routine adds a user name and password entry to the login table. Note that what is saved in the login table is the user name and the address of *passwd*, not the actual password.

The length of user names should not exceed **MAX_LOGIN_NAME_LEN**, while the length of passwords depends on the encryption routine used. For the default encryption routine, passwords should be at least 8 characters long and no more than **MAX_PASSWORD_LEN** characters.

The procedure for adding a new user to login table is as follows:

(1) Generate the encrypted password by invoking **vxencrypt** in **host/***hostOs*/bin.

(2) Add a user by invoking **loginUserAdd( )** in the VxWorks shell with the user name and the encrypted password.

The password of a user can be changed by first deleting the user entry, then adding the user entry again with the new encrypted password.

EXAMPLE

```
-> loginUserAdd "peter", "RRdRd9Qbyz"
value = 0 = 0x0
-> loginUserAdd "robin", "bSzyydqbSb"
value = 0 = 0x0
-> loginUserShow

  User Name
  =========
  peter
  robin
value = 0 = 0x0
->
```

RETURNS **OK**, or **ERROR** if the user name has already been entered, or one of the arguments is **NULL**.

ERRNO Possible errnos set by this routine include:

EINVAL
An invalid argument is passed to the routine.

S_loginLib_USER_ALREADY_EXISTS
The user name *name* is already registered.

SEE ALSO **loginLib**, **loginUserVerify( )**, **loginUserDelete( )**, **vxencrypt**

loginUserDelete( )

NAME **loginUserDelete( )** – delete a user entry from the login table

SYNOPSIS

```
STATUS loginUserDelete
    (
    char * name,   /* user name */
    char * passwd  /* user password */
    )
```

DESCRIPTION This routine deletes an entry in the login table. Both the user name and password must be specified to remove an entry from the login table.

RETURNS **OK**, or **ERROR** if the specified user or password is incorrect.

ERRNO Possible errnos set by this routine include:

EINVAL
An invalid argument is passed to the routine.

S_loginLib_UNKNOWN_USER
Unknown user name *name*.

S_loginLib_INVALID_PASSWORD
Invalid password *passwd* for *name*

Encryption routine's errnos (see **loginEncryptInstall( )**)

SEE ALSO **loginLib**, **loginUserAdd( )**

loginUserShow( )

NAME

loginUserShow( ) – display the user login table

SYNOPSIS

```
void loginUserShow (void)
```

DESCRIPTION

This routine displays valid user names.

EXAMPLE

```
    -> loginUserShow ()

      User Name
      =========
      peter
      robin
    value = 0 = 0x0
```

RETURNS

N/A

ERRNO

N/A

SEE ALSO

loginLib

loginUserVerify( )

NAME

loginUserVerify( ) – verify a user name and password in the login table

SYNOPSIS

```
STATUS loginUserVerify
    (
    char * name,    /* name of user */
    char * passwd   /* password of user */
    )
```

DESCRIPTION

This routine verifies a user entry in the login table.

RETURNS

OK, or **ERROR** if the user name or password is not found.

ERRNO

Possible errnos set by this routine include:

EINVAL
An invalid argument is passed to the routine.

S_loginLib_UNKNOWN_USER
Unknown user name *name*.

S_loginLib_INVALID_PASSWORD
Invalid password *passwd* for *name*

Encryption routine's errnos (see **loginEncryptInstall( )**)

SEE ALSO **loginLib**, **loginUserAdd( )**

logout( )

NAME **logout( )** – log out of the VxWorks system

SYNOPSIS

```
void logout (void)
```

DESCRIPTION This command logs out of the VxWorks shell. If a remote login is active (via **rlogin** or **telnet**), it is stopped, and standard I/O is restored to the console.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **rlogin( )**, **telnet( )**, **shellLogout( )**, the VxWorks programmer guides.

ls( )

NAME **ls( )** – generate a brief listing of a directory

SYNOPSIS

```
STATUS ls
    (
    const char * dirName,  /* name of dir to list */
    BOOL         doLong    /* switch on details */
    )
```

DESCRIPTION This function is simply a front-end for **dirList( )**, intended for brevity and backward compatibility. It produces a list of files and directories, without details such as file size and date, and without recursion into subdirectories.

dirName is a name of a directory or file, and may contain wildcards. *doLong* is provided for backward compatibility.

NOTE This is a target resident function, which manipulates the target I/O system. It must be preceded with the @ letter if executed from the Host Shell (windsh), which has a built-in command of the same name that operates on the Host's I/O system.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dirList( )**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

lseek( )

NAME **lseek( )** – set a file read/write pointer

SYNOPSIS

```
off_t lseek
    (
    int   fd,      /* file descriptor            */
    off_t offset,  /* new byte offset to seek to */
    int   whence   /* relative file position     */
    )
```

DESCRIPTION This routine sets the file read/write pointer of file *fd* to *offset*. The argument *whence*, which affects the file position pointer, has three values:

SEEK_SET (0)	set to *offset*
SEEK_CUR (1)	set to current position plus *offset*
SEEK_END (2)	set to the size of the file plus *offset*

This routine calls **ioctl( )** with functions FIOWHERE, FIONREAD, and FIOSEEK.

RETURNS The new offset from the beginning of the file, or **ERROR**.

ERRNO See **ioctl( )**.

SEE ALSO **ioLib**

lsr( )

NAME

lsr( ) – list the contents of a directory and any of its subdirectories

SYNOPSIS

```
STATUS lsr
    (
    const char * dirName  /* name of dir to list */
    )
```

DESCRIPTION

This function is simply a front-end for **dirList( )**, intended for brevity and backward compatibility. It produces a list of files and directories, without details such as file size and date, with recursion into subdirectories.

dirName is a name of a directory or file, and may contain wildcards.

RETURNS

OK or **ERROR**.

ERRNO

Not Available

SEE ALSO

usrFsLib, **dirList( )**, the VxWorks programmer guides.

lstAdd( )

NAME

lstAdd( ) – add a node to the end of a list

SYNOPSIS

```
void lstAdd
    (
    LIST *pList,  /* pointer to list descriptor */
    NODE *pNode   /* pointer to node to be added */
    )
```

DESCRIPTION

This routine adds a specified node to the end of a specified list.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

lstLib

lstConcat( )

NAME

lstConcat( ) – concatenate two lists

SYNOPSIS

```
void lstConcat
    (
    FAST LIST *pDstList,  /* destination list */
    FAST LIST *pAddList   /* list to be added to dstList */
    )
```

DESCRIPTION

This routine concatenates the second list to the end of the first list. The second list is left empty. Either list (or both) can be empty at the beginning of the operation.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

lstLib

lstCount( )

NAME

lstCount( ) – report the number of nodes in a list

SYNOPSIS

```
int lstCount
    (
    LIST *pList  /* pointer to list descriptor */
    )
```

DESCRIPTION

This routine returns the number of nodes in a specified list.

RETURNS

The number of nodes in the list.

ERRNO

Not Available

SEE ALSO

lstLib

lstDelete( )

NAME **lstDelete( )** – delete a specified node from a list

SYNOPSIS

```
void lstDelete
    (
    FAST LIST *pList,  /* pointer to list descriptor */
    FAST NODE *pNode   /* pointer to node to be deleted */
    )
```

DESCRIPTION This routine deletes a specified node from a specified list.

RETURNS N/A

ERRNO Not Available

SEE ALSO **lstLib**

lstExtract( )

NAME **lstExtract( )** – extract a sublist from a list

SYNOPSIS

```
void lstExtract
    (
    FAST LIST *pSrcList,    /* pointer to source list */
    FAST NODE *pStartNode,  /* first node in sublist to be extracted */
    FAST NODE *pEndNode,    /* last node in sublist to be extracted */
    FAST LIST *pDstList     /* ptr to list where to put extracted list */
    )
```

DESCRIPTION This routine extracts the sublist that starts with *pStartNode* and ends with *pEndNode* from a source list. It places the extracted list in *pDstList*.

RETURNS N/A

ERRNO Not Available

SEE ALSO **lstLib**

lstFind( )

NAME

lstFind( ) – find a node in a list

SYNOPSIS

```
int lstFind
    (
    LIST      *pList,  /* list in which to search */
    FAST NODE *pNode   /* pointer to node to search for */
    )
```

DESCRIPTION

This routine returns the node number of a specified node (the first node is 1).

RETURNS

The node number, or **ERROR** if the node is not found.

ERRNO

Not Available

SEE ALSO

lstLib

lstFirst( )

NAME

lstFirst( ) – find first node in list

SYNOPSIS

```
NODE *lstFirst
    (
    LIST *pList  /* pointer to list descriptor */
    )
```

DESCRIPTION

This routine finds the first node in a linked list.

RETURNS

A pointer to the first node in a list, or **NULL** if the list is empty.

ERRNO

Not Available

SEE ALSO

lstLib

lstFree( )

NAME **lstFree( )** – free up a list

SYNOPSIS

```
void lstFree
    (
    LIST *pList  /* list for which to free all nodes */
    )
```

DESCRIPTION This routine turns any list into an empty list. It also frees up memory used for nodes.

RETURNS N/A

ERRNO Not Available

SEE ALSO **lstLib**, **free( )**

lstGet( )

NAME **lstGet( )** – delete and return the first node from a list

SYNOPSIS

```
NODE *lstGet
    (
    FAST LIST *pList  /* ptr to list from which to get node */
    )
```

DESCRIPTION This routine gets the first node from a specified list, deletes the node from the list, and returns a pointer to the node gotten.

RETURNS A pointer to the node gotten, or **NULL** if the list is empty.

ERRNO Not Available

SEE ALSO **lstLib**

lstInit( )

NAME

lstInit( ) – initialize a list descriptor

SYNOPSIS

```
void lstInit
    (
    FAST LIST *pList  /* ptr to list descriptor to be initialized */
    )
```

DESCRIPTION

This routine initializes a specified list to an empty list.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

lstLib

lstInsert( )

NAME

lstInsert( ) – insert a node in a list after a specified node

SYNOPSIS

```
void lstInsert
    (
    FAST LIST *pList,  /* pointer to list descriptor */
    FAST NODE *pPrev,  /* pointer to node after which to insert */
    FAST NODE *pNode   /* pointer to node to be inserted */
    )
```

DESCRIPTION

This routine inserts a specified node in a specified list. The new node is placed following the list node *pPrev*. If *pPrev* is **NULL**, the node is inserted at the head of the list.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

lstLib

lstLast( )

NAME

lstLast( ) – find the last node in a list

SYNOPSIS

```
NODE *lstLast
    (
    LIST *pList  /* pointer to list descriptor */
    )
```

DESCRIPTION

This routine finds the last node in a list.

RETURNS

A pointer to the last node in the list, or **NULL** if the list is empty.

ERRNO

Not Available

SEE ALSO

lstLib

lstNStep( )

NAME

lstNStep( ) – find a list node *nStep* steps away from a specified node

SYNOPSIS

```
NODE *lstNStep
    (
    FAST NODE *pNode,  /* the known node */
    int       nStep    /* number of steps away to find */
    )
```

DESCRIPTION

This routine locates the node *nStep* steps away in either direction from a specified node. If *nStep* is positive, it steps toward the tail. If *nStep* is negative, it steps toward the head. If the number of steps is out of range, **NULL** is returned.

RETURNS

A pointer to the node *nStep* steps away, or **NULL** if the node is out of range.

ERRNO

Not Available

SEE ALSO

lstLib

lstNext( )

NAME

lstNext( ) – find the next node in a list

SYNOPSIS

```
NODE *lstNext
    (
    NODE *pNode  /* ptr to node whose successor is to be found */
    )
```

DESCRIPTION

This routine locates the node immediately following a specified node.

RETURNS

A pointer to the next node in the list, or **NULL** if there is no next node.

ERRNO

Not Available

SEE ALSO

lstLib

lstNth( )

NAME

lstNth( ) – find the Nth node in a list

SYNOPSIS

```
NODE *lstNth
    (
    FAST LIST *pList,  /* pointer to list descriptor */
    FAST int  nodenum  /* number of node to be found */
    )
```

DESCRIPTION

This routine returns a pointer to the node specified by a number *nodenum* where the first node in the list is numbered 1. Note that the search is optimized by searching forward from the beginning if the node is closer to the head, and searching back from the end if it is closer to the tail.

RETURNS

A pointer to the Nth node, or **NULL** if there is no Nth node.

ERRNO

Not Available

SEE ALSO

lstLib

lstPrevious( )

NAME

lstPrevious( ) – find the previous node in a list

SYNOPSIS

```
NODE *lstPrevious
    (
    NODE *pNode  /* ptr to node whose predecessor is to be found */
    )
```

DESCRIPTION

This routine locates the node immediately preceding the node pointed to by *pNode*.

RETURNS

A pointer to the previous node in the list, or **NULL** if there is no previous node.

ERRNO

Not Available

SEE ALSO

lstLib

m( )

NAME

m( ) – modify memory

SYNOPSIS

```
void m
    (
    void * adrs,  /* address to change */
    int    width  /* width of unit to be modified (1, 2, 4, 8) */
    )
```

DESCRIPTION

This command prompts the user for modifications to memory in byte, short word, or long word specified by *width*, starting at the specified address. It prints each address and the current contents of that address, in turn. If *adrs* or *width* is zero or absent, it defaults to the previous value.

The user can respond in one of several ways:

[RETURN]
Do not change this address, but continue, prompting at the next address.

number
Set the content of this address to *number*.

. (dot)
Do not change this address, and quit.

[EOF]
Do not change this address, and quit.

All numbers entered and displayed are in hexadecimal.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **mRegs( )**, the VxWorks programmer guides.

m6845vxbRegister( )

NAME **m6845vxbRegister( )** – register m6845vxb driver

SYNOPSIS

```
void m6845vxbRegister(void)
```

DESCRIPTION This routine registers the m6845vxb driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO **vxbM6845Vga**

m85xxCCSRRegister( )

NAME **m85xxCCSRRegister( )** – register m85xxLAWBAR driver

SYNOPSIS

```
void m85xxCCSRRegister (void)
```

DESCRIPTION This routine registers the m85xxLAWBAR driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO **m85xxCCSR**

mRegs( )

NAME **mRegs( )** – modify registers

SYNOPSIS

```
STATUS mRegs
    (
    char * regName,      /* register name, NULL for all */
    int    taskNameOrId  /* task name or task ID, 0 = default task */
    )
```

DESCRIPTION This command modifies the specified register for the specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed. If the specified register is not found, it prints out the valid register list and returns **ERROR**. If no register is specified, it sequentially prompts the user for new values for a task's registers. It displays each register and the current contents of that register, in turn. The user can respond in one of several ways:

[RETURN]
Do not change this register, but continue, prompting at the next register.

number
Set this register to *number*.

. (dot)
Do not change this register, and quit.

[**EOF**]
Do not change this register, and quit.

All numbers are entered and displayed in hexadecimal, except floating-point values, which may be entered in double precision.

RETURNS **OK**, or **ERROR** if the task or register does not exist.

ERRNO N/A

SEE ALSO **usrLib**, **m( )**, the VxWorks programmer guides.

mach( )

NAME

mach( ) – return the contents of system register **mach** (also **macl**, **pr**) (SH)

SYNOPSIS

```
int mach
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register mach from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for other system registers (**macl**, **pr**): **macl( )**, **pr( )**. Note that **pc( )** is provided by **usrLib.c**.

RETURNS

The contents of register mach (or the requested system register).

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

malloc( )

NAME

malloc( ) – allocate a block of memory from the system memory partition (ANSI)

SYNOPSIS

```
void * malloc
    (
    size_t nBytes  /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a block of memory from the free lists of the system memory partition (kernel heap). The size of the block will be equal to or greater than *nBytes*.

RETURNS

A pointer to the allocated block of memory, or a null pointer if there is an error.

ERRNO

Possible errnos generated by this routine include:

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO

memPartLib, **free( )**, **calloc( )**, **valloc( )**, **memPartAlloc( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: General Utilities* (**stdlib.h**)

memAddToPool( )

NAME **memAddToPool( )** – add memory to the system memory partition

SYNOPSIS

```
STATUS memAddToPool
    (
    FAST char *   pPool,    /* pointer to memory block */
    FAST unsigned poolSize  /* block size in bytes */
    )
```

DESCRIPTION This routine adds memory to the system memory partition (kernel heap), in addition to the amount of memory specified during its creation.

RETURNS **OK** or **ERROR**.

ERRNO Possible errnos generated by this routine include:

S_memLib_INVALID_ADDRESS
pPool is equal to **NULL**.

S_memLib_INVALID_NBYTES
poolSize value is too small.

SEE ALSO **memPartLib**, **memPartAddToPool( )**

memDevCreate( )

NAME **memDevCreate( )** – create a memory device

SYNOPSIS

```
STATUS memDevCreate
    (
    char * name,   /* device name              */
    char * base,   /* where to start in memory */
    int    length  /* number of bytes          */
    )
```

DESCRIPTION This routine creates a memory device containing a single file. Memory for the device is simply an absolute memory location beginning at *base*. The *length* parameter indicates the size of memory.

For example, to create the device "/mem/cpu0/", a device for accessing the entire memory of the local processor, the proper call would be:

```
    memDevCreate ("/mem/cpu0/", 0, sysMemTop())
```

The device is created with the specified name, start location, and size.

To open a file descriptor to the memory, use **open( )**. Specify a pseudo-file name of the byte offset desired, or open the "raw" file at the beginning and specify a position to seek to. For example, the following call to **open( )** allows memory to be read starting at decimal offset 1000.

```
    -> fd = open ("/mem/cpu0/1000", O_RDONLY, 0)
```

Pseudo-file name offsets are scanned with "%d".

CAVEAT The FIOSEEK operation overrides the offset given via the pseudo-file name at open time.

EXAMPLE Consider a system configured with two CPUs in the backplane and a separate dual-ported memory board, each with 1 megabyte of memory. The first CPU is mapped at VMEbus address 0x00400000 (4 Meg.), the second at bus address 0x00800000 (8 Meg.), the dual-ported memory board at 0x00c00000 (12 Meg.). Three devices can be created on each CPU as follows. On processor 0:

```
    -> memDevCreate ("/mem/local/", 0, sysMemTop())
    ...
    -> memDevCreate ("/mem/cpu1/", 0x00800000, 0x00100000)
    ...
    -> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

On processor 1:

```
    -> memDevCreate ("/mem/local/", 0, sysMemTop())
    ...
    -> memDevCreate ("/mem/cpu0/", 0x00400000, 0x00100000)
    ...
    -> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

Processor 0 has a local disk. Data or an object module needs to be passed from processor 0 to processor 1. To accomplish this, processor 0 first calls:

```
    -> copy </disk1/module.o >/mem/share/0
```

Processor 1 can then be given the load command:

```
    -> ld </mem/share/0
```

RETURNS **OK**, or **ERROR** if memory is insufficient or the I/O system cannot add the device.

ERRNO **S_ioLib_NO_DRIVER**

SEE ALSO **memDrv**

memDevCreateDir( )

NAME

memDevCreateDir( ) – create a memory device for multiple files

SYNOPSIS

```
STATUS memDevCreateDir
    (
    char *              name,     /* device name                */
    MEM_DRV_DIRENTRY * files,     /* array of dir. entries - not copied */
    int                 numFiles  /* number of entries          */
    )
```

DESCRIPTION

This routine creates a memory device for a collection of files organised into directories. The given array of directory entry records describes a number of files, some of which may be directories, represented by their own directory entry arrays. The structure may be arbitrarily deep. This effectively allows a filesystem to be created and installed in VxWorks, for essentially read-only use. The filesystem structure can be created on the host using the memdrvbuild utility.

Note that the array supplied is not copied; a reference to it is kept. This array should not be modified after being passed to memDevCreateDir.

RETURNS

OK, or **ERROR** if memory is insufficient or the I/O system cannot add the device.

ERRNO

S_ioLib_NO_DRIVER

SEE ALSO

memDrv

memDevDelete( )

NAME

memDevDelete( ) – delete a memory device

SYNOPSIS

```
STATUS memDevDelete
    (
    char * name  /* device name */
    )
```

DESCRIPTION

This routine deletes a memory device containing a single file or a collection of files. The device is deleted with it own name.

For example, to delete the device created by memDevCreate ("/mem/cpu0/", 0, **sysMemTop( )**), the proper call would be:

```
 memDevDelete ("/mem/cpu0/");
```

RETURNS **OK**, or **ERROR** if the device doesn't exist.

ERRNO N/A.

SEE ALSO **memDrv**

memDrv( )

NAME **memDrv( )** – install a memory driver

SYNOPSIS

```
STATUS memDrv (void)
```

DESCRIPTION This routine initializes the memory driver. It is called automatically when VxWorks is configured with the **INCLUDE_MEMDRV** component.

RETURNS **OK**, or **ERROR** if the I/O system cannot install the driver.

ERRNO N/A.

SEE ALSO **memDrv**

memEdrBlockMark( )

NAME **memEdrBlockMark( )** – mark or unmark selected blocks

SYNOPSIS

```
int memEdrBlockMark
    (
    int  partId,  /* partition ID selector */
    int  taskId,  /* task ID selector */
    BOOL unmark   /* TRUE to unmark */
    )
```

DESCRIPTION This routine marks blocks selected by partition ID and/or taskId. Passing **NULL** for either *partId* or *taskId* means no filtering is done for that field.

RETURNS number of newly marked or unmarked blocks

ERRNO Not Available

SEE ALSO **memEdrLib**, **memEdrBlockShow( )**

memEdrBlockShow( )

NAME

memEdrBlockShow( ) – print memory block information

SYNOPSIS

```
STATUS memEdrBlockShow
    (
    int    partId,     /* partition ID selector */
    void * addr,       /* address selector */
    int    taskId,     /* task ID selector */
    UINT   type,       /* block type selector */
    UINT   level,      /* detail level */
    BOOL   continuous  /* print in continuous mode */
    )
```

DESCRIPTION

This routine displays memory block information based on various selection criteria. **NULL** or 0 can be used for *partId*, *addr*, *taskId* and *type* to exclude the respective field from filtering. The *level* parameter can be used to enable printing of extended block information (trace) when collection of extended information is enabled.

The following *type* parameters are accepted:

type	description
0	any block
1	global variable reported by RTC
2	allocated block
3	queued free block
4	marked allocated block
5	unmarked allocated block

If more than 20 blocks match the selection criteria and *continuous* is **FALSE**, blocks are printed 20 at a time. With continuous mode information is also collected 20 at a time, but printing is continuous, with no user intervention enabled. Note that either way, after each batch of 20 blocks the mutex lock is released, allowing other tasks to change the instrumentation database.

RETURNS

OK, or **ERROR** if getting the info failed.

ERRNO

N/A

SEE ALSO

memEdrShow, **memEdrLib**, **memEdrPartShow( )**, **memEdrBlockMark( )**

memEdrFreeQueueFlush( )

NAME **memEdrFreeQueueFlush( )** – flush the free queue

SYNOPSIS

```
void memEdrFreeQueueFlush (void)
```

DESCRIPTION This routine can be used to remove all blocks queued on the free queue, and finalize the free operation. This way memory blocks previously queued will be freed into their respective memory partitions.

RETURNS N/A

ERRNO Not Available

SEE ALSO **memEdrLib**

memEdrPartShow( )

NAME **memEdrPartShow( )** – show partition information in the kernel

SYNOPSIS

```
STATUS memEdrPartShow
    (
    PART_ID partId  /* partition ID */
    )
```

DESCRIPTION This routine displays information about memory partitions in the kernel. If the *partId* parameter is **NULL**, it lists all partitions recorded in the kernel's database.

RETURNS **OK**, or **ERROR** if getting the info failed.

ERRNO N/A

SEE ALSO **memEdrShow**, **memEdrLib**, **memEdrBlockShow( )**

memEdrRtpBlockMark( )

NAME

memEdrRtpBlockMark( ) – mark or unmark selected allocated blocks in an RTP

SYNOPSIS

```
int memEdrRtpBlockMark
    (
    RTP_ID rtpId,   /* RTP id */
    int    partId,  /* partition ID selector */
    int    taskId,  /* task ID selector */
    BOOL   unmark   /* TRUE to unmark */
    )
```

DESCRIPTION

This routine marks blocks selected by partition ID and/or task ID. Passing **NULL** for either *partId* or *taskId* means no filtering is done using that field.

This routine only works with RTPs with the memory manager instrumentation (**memEdrLib**) enabled and the **MEDR_SHOW_ENABLE** environment variable set to **TRUE**.

RETURNS

number of newly marked or unmarked blocks

ERRNO

N/A

SEE ALSO

memEdrRtpShow, **memEdrLib**, **memEdrRtpBlockShow( )**

memEdrRtpBlockShow( )

NAME

memEdrRtpBlockShow( ) – print memory block information of an RTP

SYNOPSIS

```
STATUS memEdrRtpBlockShow
    (
    RTP_ID rtpId,      /* RTP id */
    int    partId,     /* partition ID selector */
    void * addr,       /* address selector */
    int    taskId,     /* task ID selector */
    UINT   type,       /* block type selector */
    UINT   level,      /* detail level */
    BOOL   continuous  /* print in continuous mode */
    )
```

DESCRIPTION

This routine displays memory block information based on various selection criteria. **NULL** or 0 can be used for *partId*, *addr*, *taskId* and *type* to exclude the respective field from filtering. The *level* parameter can be used to enable printing of extended block information (trace) when collection of extended information is enabled.

The following *type* parameters are accepted:

type	description
0	any block
1	global variable reported by RTC
2	allocated block (marked or unmarked)
3	queued free block
4	marked allocated block
5	unmarked allocated block

If more than 20 blocks match the selection criteria and *continuous* is **FALSE**, blocks are printed 20 at a time. With continuous mode information is also collected 20 at a time, but printing is continuous, with no user intervention enabled. Note that between each batch of 20 blocks the mutex lock is released allowing other tasks to change the instrumentation database.

This routine only works with RTPs with the memory manager instrumentation (**memEdrLib**) enabled and the **MEDR_SHOW_ENABLE** environment variable set to **TRUE**. For symbolic information, the RTP has to be spawned with the **RTP_WITH_SYMBOLS** option.

RETURNS **OK**, or **ERROR** if getting the info failed.

ERRNO N/A

SEE ALSO **memEdrRtpShow**, **memEdrLib**, **memEdrRtpPartShow( )**, **memEdrRtpBlockMark( )**

memEdrRtpPartShow( )

NAME **memEdrRtpPartShow( )** – show partition information of an RTP

SYNOPSIS

```
STATUS memEdrRtpPartShow
    (
    RTP_ID rtpId,  /* RTP id */
    int    partId  /* partition ID selector */
    )
```

DESCRIPTION This routine displays information about memory partitions in an RTP. If the *partId* parameter is **NULL**, it lists all partitions recorded in the RTP's database.

This routine only works with RTPs with the memory manager instrumentation (**memEdrLib**) enabled and the **MEDR_SHOW_ENABLE** environment variable set to **TRUE**.

RETURNS **OK**, or **ERROR** if getting the info failed.

ERRNO N/A

SEE ALSO **memEdrRtpShow**, **memEdrLib**, **memEdrRtpBlockShow( )**

memFindMax( )

NAME **memFindMax( )** – find the largest free block in the system memory partition (kernel heap)

SYNOPSIS

```
int memFindMax (void)
```

DESCRIPTION This routine searches for the largest block in the system memory partition (kernel heap) free list and returns its size. It returns 0 if there is no free block in the system memory partition. The size returned corresponds to the largest block that can be allocated using the default alignment value, which is used via calls to **malloc( )**, **realloc( )**, or **calloc( )**. Allocation of such a size with an alignment greater than the default aligment will fail: this may occur when using **memalign( )** or **valloc( )**. The default alignment value is documented in the manual entry for **memPartLib** as the architecture specific **boundary**.

RETURNS The size, in bytes, of the largest available block.

ERRNO Not Available

SEE ALSO **memInfo**, **memPartFindMax( )**

memInfoGet( )

NAME **memInfoGet( )** – get heap information

SYNOPSIS

```
STATUS memInfoGet
    (
    MEM_PART_STATS * pPartStats  /* partition stats structure */
    )
```

DESCRIPTION This routine takes a pointer to a **MEM_PART_STATS** structure. All fields of the structure are filled in with data from the RTP heap memory partition. For the description of the information provided, see the **memPartInfoGet( )** documentation.

RETURNS **OK** if the structure has valid data, otherwise **ERROR**.

ERRNO Not Available

SEE ALSO **memInfo**, **memPartInfoGet( )**

memOptionsGet( )

NAME **memOptionsGet( )** – get the options of the system memory partition (kernel heap)

SYNOPSIS

```
STATUS memOptionsGet
    (
    UINT * pOptions  /* pointer to options for kernel heap */
    )
```

DESCRIPTION This routine sets the parameter *pOptions* with the options of the system memory partition (kernel heap).

Heap/memory partition options are discussed in details in the reference entry for the library **memLib**.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **memLib**, **memOptionsSet( )**, **memPartOptionsGet( )**, **memPartOptionsSet( )**

memOptionsSet( )

NAME **memOptionsSet( )** – set the options for the system memory partition (kernel heap)

SYNOPSIS

```
STATUS memOptionsSet
    (
    unsigned options  /* options for system memory partition (kernel heap) */
    )
```

DESCRIPTION This routine sets the debug and error handling options for the system memory partition (kernel heap). For detailed description of these options see the **memLib** and **memPartOptionsSet( )**.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **memLib**, **memOptionsGet( )**, **memPartOptionsSet( )**, **memPartOptionsGet( )**

memPartAddToPool( )

NAME **memPartAddToPool( )** – add memory to a memory partition

SYNOPSIS

```
STATUS memPartAddToPool
    (
    FAST PART_ID  partId,   /* partition to add memory to */
    FAST char *   pPool,    /* pointer to memory block */
    FAST unsigned poolSize  /* block size in bytes */
    )
```

DESCRIPTION This routine adds memory to a specified memory partition already created with **memPartCreate( )**. The memory added need not be contiguous with memory previously assigned to the partition.

The size of the memory pool being added has to be large enough to accommodate the section overhead consisting of a section header and some reserved blocks that mark the beginning and the end of the section. This overhead, approximately 64 bytes, is not available for allocation.

This routine does not verify that the memory block passed corresponds to valid memory or not. It is the user's responsability to ensure that the block is valid and it does not overlap with other blocks added to the partition.

RETURNS **OK** or **ERROR**.

ERRNO Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

S_memLib_INVALID_ADDRESS
pPool is equal to **NULL**.

S_memLib_INVALID_NBYTES
poolSize value is too small.

SEE ALSO **memPartLib**, **smMemLib**, **memPartCreate( )**, **memAddToPool( )**

memPartAlignedAlloc( )

NAME

memPartAlignedAlloc( ) – allocate aligned memory from a partition

SYNOPSIS

```
void * memPartAlignedAlloc
    (
    FAST PART_ID partId,    /* memory partition to allocate from */
    unsigned     nBytes,    /* number of bytes to allocate */
    unsigned     alignment  /* boundary to align to */
    )
```

DESCRIPTION

This routine allocates a buffer of size *nBytes* from a specified partition. Additionally, it ensures that the allocated buffer begins on a memory address evenly divisible by *alignment*. The *alignment* parameter must be a power of 2.

RETURNS

A pointer to the newly allocated block, or **NULL** if the buffer could not be allocated.

ERRNO

Possible errnos generated by this routine include:

S_memLib_INVALID_ALIGNMENT
alignment is not a power of two.

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO

memPartLib, **memalign( )**

memPartAlloc( )

NAME

memPartAlloc( ) – allocate a block of memory from a partition

SYNOPSIS

```
void * memPartAlloc
    (
    FAST PART_ID partId,  /* memory partition to allocate from */
    unsigned     nBytes   /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a block of memory from a specified partition. The size of the block will be equal to or greater than *nBytes*. The partition must already be created with **memPartCreate( )**.

RETURNS

A pointer to a block, or **NULL** if the call fails.

ERRNO

Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO **memPartLib**, **smMemLib**, **memPartCreate( )**, **malloc( )**

memPartCreate( )

NAME **memPartCreate( )** – create a memory partition

SYNOPSIS

```
PART_ID memPartCreate
    (
    char *   pPool,     /* pointer to memory area */
    unsigned poolSize   /* size in bytes */
    )
```

DESCRIPTION This routine creates a new memory partition containing a specified memory pool defined by its start address, *pPool*, and its size in bytes, *poolSize*. It returns a partition ID, which can be passed to other routines to manage the partition (i.e., to allocate and free memory blocks in the partition). Partitions can be created to manage any number of separate memory pools.

Empty memory partitions can be created by setting *pPool* to **NULL** and *poolSize* to 0. For such partitions, it is necessary to add memory blocks to the partition via **memPartAddToPool( )** before performing any allocation request.

Unless creating an empty partition, the memory pool size has to be large enough to accomodate some overhead consisting of a section header and some reserved blocks that mark the beginning and the end of the section. In addition, certain internal data structures used to store free block information are also carved from the pool. This overhead, in total approximately 248 bytes, is not available for allocations.

The create routine does not verify that the memory block passed corresponds to valid memory or not. It is the user's responsability to make sure the block is valid.

NOTE The descriptor for the new partition object is allocated out of the system memory partition (i.e., with **malloc( )**).

RETURNS The partition ID, or **NULL** if there is insufficient memory in the system memory partition (kernel heap) for a new partition descriptor, or *poolSize* value is too small.

ERRNO Possible errnos generated by this routine include:

S_memLib_INVALID_NBYTES
poolSize value is too small.

SEE ALSO **memPartLib**, **smMemLib**

memPartDelete( )

NAME **memPartDelete( )** – delete a partition and free associated memory

SYNOPSIS

```
STATUS memPartDelete
    (
    PART_ID partId  /* partition to delete */
    )
```

DESCRIPTION This routine deletes the memory partition object. It is supported for local memory partition but not for shared memory partition.

RETURNS **OK** or **ERROR**.

ERRNO Possible errnos generated by this routine include:

S_memLib_NO_PARTITION_DESTROY
feature not supported for shared memory partition.

SEE ALSO **memPartLib**

memPartFindMax( )

NAME **memPartFindMax( )** – find the size of the largest available free block

SYNOPSIS

```
int memPartFindMax
    (
    FAST PART_ID partId  /* partition ID */
    )
```

DESCRIPTION This routine searches for the largest block in the memory partition free list and returns its size. It returns 0 if there is no free block in the memory partition. The size returned corresponds to the largest block that can be allocated using the default alignment value, which is used via calls to **memPartAlloc( )**, or **memPartRealloc( )**. Allocation of such a size with an alignment greater than the default aligment will fail: this may occur when using

memPartAlignedAlloc( ). The default alignment value is documented in the manual entry for **memPartLib** as the architecture specific **boundary**.

RETURNS The size, in bytes, of the largest available block.

ERRNO Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

SEE ALSO **memInfo**, **smMemLib**, **memFindMax( )**

memPartFree( )

NAME **memPartFree( )** – free a block of memory in a partition

SYNOPSIS

```
STATUS memPartFree
    (
    PART_ID partId,  /* memory partition to free a block from */
    char *  pBlock   /* pointer to block of memory to free */
    )
```

DESCRIPTION This routine returns to a partition's free memory lists a block of memory previously allocated with **memPartAlloc( )**, **memPartAlignedAlloc( )** or **memPartRealloc( )**. If *pBlock* is a null pointer, no action occurs and the function returns **OK**.

RETURNS **OK**, or **ERROR** if the block or the partition is invalid.

ERRNO Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

S_memLib_WRONG_PART_ID
The block does not belong to the partition.

SEE ALSO **memPartLib**, **smMemLib**, **memPartAlloc( )**, **memPartAlignedAlloc( )**, **free( )**

memPartInfoGet( )

NAME

memPartInfoGet( ) – get partition information

SYNOPSIS

```
STATUS memPartInfoGet
    (
    PART_ID             partId,     /* partition ID              */
    MEM_PART_STATS    * pPartStats  /* partition stats structure */
    )
```

DESCRIPTION

This routine takes a partition ID and a pointer to a **MEM_PART_STATS** structure. All the parameters of the structure are filled in with the current partition information which include:

numBytesFree
number of free bytes in the partition

numBlocksFree
number of free blocks in the partition

maxBlockSizeFree
maximum block size in bytes that is free

numBytesAlloc
number of allocated bytes in the partition

numBlocksAlloc
number of allocated blocks in the partition

maxBytesAlloc
maximum number of allocated bytes at any time (peak usage)

RETURNS

OK if the structure has valid data, otherwise **ERROR**.

ERRNO

Not Available

SEE ALSO

memInfo, **memShow( )**, **memPartShow( )**

memPartOptionsGet( )

NAME

memPartOptionsGet( ) – get the options of a memory partition

SYNOPSIS

```
STATUS memPartOptionsGet
    (
    PART_ID partId,   /* partition to set option for  */
    UINT *  pOptions  /* pointer to partition options */
    )
```

DESCRIPTION This routine sets the parameter *pOptions* with the options of a specified memory partition.

RETURNS **OK**, or **ERROR** if partition is shared or *pOptions* is a **NULL** pointer.

ERRNO Possible errnos generated by this routine include:

S_memLib_FUNC_NOT_AVAILABLE
partId is a shared partition for which **memPartOptionsGet( )** is not supported.

SEE ALSO **memLib**, **smMemLib**, **memPartOptionsSet( )**, **memOptionsGet( )**

memPartOptionsSet( )

NAME **memPartOptionsSet( )** – set the options for a memory partition

SYNOPSIS

```
STATUS memPartOptionsSet
    (
    PART_ID  partId,  /* partition to set option for */
    unsigned options  /* memory management options */
    )
```

DESCRIPTION This routine sets the debug options for a specified memory partition. 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, the error status is returned. For the supported options see the **memLib** library reference guide.

RETURNS **OK** or **ERROR**.

ERRNO Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

SEE ALSO **memLib**, **smMemLib**, **memPartOptionsGet( )**, **memOptionsSet( )**

memPartRealloc( )

NAME

memPartRealloc( ) – reallocate a block of memory in a specified partition

SYNOPSIS

```
void * memPartRealloc
    (
    PART_ID  partId,  /* partition ID */
    char *   pBlock,  /* block to be reallocated */
    unsigned nBytes   /* new block size in bytes */
    )
```

DESCRIPTION

This routine changes the size of a specified block of memory and returns a pointer to the new block. The contents that fit inside the new size (or old size if smaller) remain unchanged. The memory alignment of the new block is not guaranteed to be the same as the original block.

If *pBlock* is **NULL**, this call is equivalent to **memPartAlloc( )**.

If *nBytes* is set to zero and *pBlock* points to a valid allocated block, this call is equivalent to **memPartFree( )** and returns **NULL**.

RETURNS

A pointer to the new block of memory, **NULL** if the call fails or *nBytes* is equal to zero.

ERRNO

Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

S_memLib_WRONG_PART_ID
The block does not belong to the partition.

SEE ALSO

memLib, **smMemLib**, **realloc( )**

memPartShow( )

NAME

memPartShow( ) – show blocks and statistics for a given memory partition

SYNOPSIS

```
STATUS memPartShow
```

```
    (
    PART_ID partId,  /* memory partition ID                    */
    int     type     /* 0 = statistics, 1 = statistics & list */
                     /* 2 = statistics & list & extra info    */
    )
```

DESCRIPTION This routine displays statistics about the available and allocated memory in a specified memory partition. For details about usage and information shown by this routine refer to the **memShow( )** documentation.

RETURNS **OK** or **ERROR**.

ERRNO Possible errnos generated by this routine include:

S_smObjLib_NOT_INITIALIZED
partId is a shared partition but the Shared Memory Allocator component was not initialized.

SEE ALSO **memShow**, **memShow( )**, **memPartAddToPool( )**

memPartSmCreate( )

NAME **memPartSmCreate( )** – create a shared memory partition (VxMP Option)

SYNOPSIS

```
PART_ID memPartSmCreate
    (
    char *   pPool,    /* global address of shared memory area */
    unsigned poolSize  /* size in bytes */
    )
```

DESCRIPTION This routine creates a shared memory partition that can be used by tasks on all CPUs in the system. It returns a partition ID which can then be passed to generic **memPartLib** routines to manage the partition (i.e., to allocate and free memory blocks in the partition).

pPool is the global address of shared memory dedicated to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory dedicated to the partition.

Before this routine can be called, the shared memory objects facility must be initialized (see **smMemLib**).

NOTE The descriptor for the new partition is allocated out of an internal dedicated shared memory partition. The maximum number of partitions that can be created is **SM_OBJ_MAX_MEM_PART** .

Memory pool size is rounded down to a 16-byte boundary.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The partition ID, or **NULL** if there is insufficient memory in the dedicated partition for a new partition descriptor.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**, **memLib**

memShow( )

NAME **memShow( )** – show blocks and statistics for the current heap partition

SYNOPSIS

```
STATUS memShow
    (
    int type  /* 0 = summary, 1 = list all blocks in the free list, */
              /* 2 = list all sections */
    )
```

DESCRIPTION This routine displays statistics about the available and allocated memory in the current heap partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, as well as the maximum block size of free memory. It also shows the number of blocks currently and cumulatively allocated, the average allocated block size and the maximum number of bytes allocated at any given time (peak usage). The cumulative information wraps around after reaching **UINT_MAX** (4GB). Part of the heap space is used internally by the partition for its bookkeeping. This amount of memory is displayed as part of the current usage.

The **memShow( )** routine called with *type* 1 or 2 requires that certain internal data structures (binary tree, linked list) are traversed. In case of corrupted internal heap structures an exception may occur, causing the task executing **memShow( )** to get suspended while holding the partition's mutex semaphore. This, in effect, results in the partition being locked indefinitely.

The **memShow( )** routine temporarily saves free block information on the stack in order to avoid performing IO while the partition semaphore is taken. To avoid excessive stack requirement, the number of lines listed in the free block section is restricted; however, at least 200 lines are printed. Sections listed are also limited to 50. The user should make sure that **memShow( )** is called in the context of a task that has sufficient stack space (approximately 4k needed for **memShow( )**).

EXAMPLE

```
-> memShow
  status       bytes       blocks   avg block  max block
 -------- ------------- ---------- ---------- ----------
current
 free          2330696         11     211881   1962688
 alloc         1858696       4961        374         -
 internal          400          2        200         -
cumulative
 alloc         2240928       5419        413         -
peak
 alloc         2202992          -          -         -
```

memShow( ) can be used to detect memory leaks within the current heap. This can be achieved by comparing values of the current number of allocated bytes before and after the function call(s) you want to verify. The current amount of free memory cannot be used to detect memory leaks since it is updated everytime that memory blocks are allocated internally by the system for the heap partition bookkeeping.

In addition, if *type* is 1, the routine displays a list of all different size of free blocks present in the heap partition. The size corresponds to the amount of usable data plus the overhead required for the block header. The heap partition options are also displayed.

EXAMPLE

```
-> memShow 1

LIST OF FREE BLOCKS:
         number      size
        --------  ----------
               1          24
               1          72
               1         104
               1         144
               1         176
               1         232
               1         264
               1        1440
               1      127080
               1      238472
               1     1962688


OPTIONS:
        ALLOC_ERROR_LOG
        BLOCK_ERROR_LOG
        BLOCK_ERROR_EDR

SUMMARY:
  status       bytes       blocks   avg block  max block
 -------- ------------- ---------- ---------- ----------
current
 free          2330696         11     211881   1962688
 alloc         1858696       4961        374         -
 internal          400          2        200         -
cumulative
 alloc         2240928       5419        413         -
```

```
    peak
     alloc          2202992           -           -          -
```

If *type* is 2, the routine also displays the address of each free block. The address of the free blocks is the start address of the free block header, not the start address of the usable data. In addition, a list of all the memory sections that were added to the heap partition with **memPartAddToPool( )** or **memAddToPool( )** is displayed.

```
    -> memShow 2

    LIST OF FREE BLOCKS:
             number      size        addr
            --------  ----------  ----------
                   1          24  0x002075f0
                   1          72  0x002b6dd0
                   1         104  0x00279a58
                   1         144  0x002b5968
                   1         176  0x0027a048
                   1         232  0x00206110
                   1         264  0x0027a1a8
                   1        1440  0x00204c58
                   1      127080  0x002b6f28
                   1      238472  0x0027a2f0
                   1     1962688  0x00420d30


    LIST OF MEMORY SECTIONS ADDED TO THE PARTITION:
            start addr     size
            ----------  ----------
            0x00206f60     4165792
            0x002003e8       20000
            0x00205208        4096

    OPTIONS:
            ALLOC_ERROR_LOG
            BLOCK_ERROR_LOG
            BLOCK_ERROR_EDR

    SUMMARY:
      status       bytes       blocks   avg block  max block
     -------- ------------- ---------- ---------- ----------
    current
     free           2330696         11     211881    1962688
     alloc          1858696       4961        374          -
     internal           400          2        200          -
    cumulative
     alloc          2240928       5419        413          -
    peak
     alloc          2202992          -          -          -
```

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **memShow**, **memPartShow( )**, **memAddToPool( )**

memShowInit( )

NAME **memShowInit( )** – initialize the memory partition show facility

SYNOPSIS

```
void memShowInit (void)
```

DESCRIPTION This routine links the memory partition show facility into the VxWorks system. These routines are included automatically when this show facility is configured into VxWorks using the **INCLUDE_MEM_SHOW** component.

RETURNS N/A

ERRNO Not Available

SEE ALSO **memShow**

memalign( )

NAME **memalign( )** – allocate aligned memory from system memory partition (kernel heap)

SYNOPSIS

```
void * memalign
    (
    unsigned alignment,  /* boundary to align to (power of 2) */
    unsigned size        /* number of bytes to allocate */
    )
```

DESCRIPTION This routine allocates a buffer of size *size* from the system memory partition (kernel heap). Additionally, it insures that the allocated buffer begins on a memory address evenly divisible by the specified alignment parameter. The alignment parameter must be a power of 2.

RETURNS A pointer to the newly allocated block, or **NULL** if the buffer could not be allocated.

ERRNO Possible errnos generated by this routine include:

S_memLib_INVALID_ALIGNMENT
alignment is not a power of two.

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO **memLib**, **memPartAlignedAlloc( )**

miiBusCreate( )

NAME **miiBusCreate( )** – create an miiBus attached to a parent bridge

SYNOPSIS

```
STATUS miiBusCreate
    (
    VXB_DEVICE_ID pDev,
    VXB_DEVICE_ID *pBus
    )
```

DESCRIPTION This function allocates a new VxBus instance and configures it to be an miiBus device. The new miiBus device inherits almost all of its properties from its parent bridge device, specified by pDev. An miiBus must be created by all ethernet device instances that have an MII-based transceiver attached, regardless of whether or not the ethernet controller itself has access to the PHY's management registers. For example, with MPC8260 boards, there are two FCC 10/100 ethernet ports which use MII PHYs, but the MDIO pins for the PHYs are typically connected to parallel I/O port D, which is logically distinct from either FCC controller. Nevertheless, FCC1 and FCC2 must both have a child MII bus, even though initially these buses will appear empty. A separate driver is required to provide a 3rd MII bus instance attached to parellel I/O port D in order to actually make the PHYs available to the system. These PHYs should be remapped to the MII buses attached to the FCC ports via remapping entries in **hwconf.c**.

The **miiBusCreate( )** function will allocate a private pDrvCtrl structure for the bus device, along with an empty **END_MEDIALIST** list. (This list will be filled in as PHYs are attached.) Assuming the new instance is successfully allocated and configured, it will then be announced to VxBus. (Note that this just tells VxBus of the new device instance. The call to **vxbBusAnnounce( )**, which tells VxBus to create a new bus instance, has to be done later.)

If the caller provided a non-**NULL** pBus pointer, it will be used to return a pointer to the newly created device to the caller.

RETURNS **OK** if a bus is successfully created, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **miiBus**

miiBusDelete( )

NAME

miiBusDelete( ) – delete an miiBus and all its child devices

SYNOPSIS

```
STATUS miiBusDelete
    (
    VXB_DEVICE_ID pDev
    )
```

DESCRIPTION

This routine is used to shut down an miiBus. All child PHY instances attached to the bus are deleted, and then the bus instance itself is destroyed. This routine should only be called by the parent driver that also called **miiBusCreate( )**.

RETURNS

OK if a bus is successfully destroyed, or **ERROR** otherwise

ERRNO

N/A

SEE ALSO

miiBus

miiBusGet( )

NAME

miiBusGet( ) – get the miiBus that goes with a given VxBus instance

SYNOPSIS

```
STATUS miiBusGet
    (
    VXB_DEVICE_ID pDev,
    VXB_DEVICE_ID *pBus
    )
```

DESCRIPTION

This routine is meant for use by ethernet drivers to locate their child miiBus instances. Normally, each ethernet driver will create an miiBus with **miiBusCreate( )**, and it should save a pointer to the instance that **miiBusCreate( )** returns (in which case it doesn't need to use this function). However, hEnd drivers currently must call **miiBusCreate( )** before their private pDrvCtrl structures are allocated, and hence have nowhere to store the miiBus pointer that **miiBusCreate( )** returns. These drivers can therefore use **miiBusGet( )** to recover and save this pointer later, usually during their **DLInit( )** routines.

RETURNS

OK if a bus is found, or **ERROR** otherwise

ERRNO

N/A

SEE ALSO

miiBus

miiBusListAdd( )

NAME

miiBusListAdd( ) – Add a PHY to the MII monitor list

SYNOPSIS

```
void miiBusListAdd
    (
    VXB_DEVICE_ID pDev
    )
```

DESCRIPTION

This function adds a PHY instance to the monitor list so that it can be checked periodically by the monitor task. This function is usually called by a PHY in its VxBus instConnect routine.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

miiBus

miiBusListDel( )

NAME

miiBusListDel( ) – Remove a PHY to the MII monitor list

SYNOPSIS

```
void miiBusListDel
    (
    VXB_DEVICE_ID pDev
    )
```

DESCRIPTION

This function removes a PHY instance from the monitor list.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

miiBus

miiBusMediaAdd( )

NAME

miiBusMediaAdd( ) – add an entry to an miiBus's media list

SYNOPSIS

```
STATUS miiBusMediaAdd
    (
    VXB_DEVICE_ID pDev,
    UINT32        media
    )
```

DESCRIPTION

This routine is used by a PHY instance to announce support for a given media type to its parent miiBus.

If the media type already exists in the list, this routine leaves the list alone and returns **ERROR**.

RETURNS

OK if media isn't a duplicate, otherwise **ERROR**

ERRNO

N/A

SEE ALSO

miiBus

miiBusMediaDefaultSet( )

NAME

miiBusMediaDefaultSet( ) – set the default media for an miiBus

SYNOPSIS

```
STATUS miiBusMediaDefaultSet
    (
    VXB_DEVICE_ID pDev,
    UINT32        media
    )
```

DESCRIPTION

This routine is used to specify which media type to specify in the endMediaListDefault member of a bus's media list. PHY driver detaches.

If the media type doesn't exist in the list, this routine returns **ERROR**.

RETURNS

OK if media entry is found in the list, otherwise **ERROR**

ERRNO

N/A

SEE ALSO

miiBus

miiBusMediaDel( )

NAME

miiBusMediaDel( ) – delete an entry to an miiBus's media list

SYNOPSIS

```
STATUS miiBusMediaDel
    (
    VXB_DEVICE_ID pDev,
    UINT32        media
    )
```

DESCRIPTION

This routine is used by a PHY instance to remove an entry for a given media type from its parent miiBus. This is used when a PHY driver detaches.

If the media type doesn't exist in the list, this routine leaves the list alone and returns **ERROR**.

RETURNS

OK if media entry is found in the list, otherwise **ERROR**

ERRNO

N/A

SEE ALSO

miiBus

miiBusMediaListGet( )

NAME

miiBusMediaListGet( ) – obtain a pointer to the bus's media list

SYNOPSIS

```
STATUS miiBusMediaListGet
    (
    VXB_DEVICE_ID    pDev,
    END_MEDIALIST ** mediaList
    )
```

DESCRIPTION

This routine returns a pointer to the bus's **END_MEDIALIST** structure to the caller. This is used by ethernet drivers that support ifmedia to service the EIOCGIFMEDIALIST ioctl.

RETURNS

OK if media list is found, or **ERROR** otherwise

ERRNO

N/A

SEE ALSO

miiBus

miiBusMediaUpdate( )

NAME

miiBusMediaUpdate( ) – invoke a PHY's parent's media update callback

SYNOPSIS

```
STATUS miiBusMediaUpdate
    (
    VXB_DEVICE_ID pDev
    )
```

DESCRIPTION

This function is used to notify the parent ethernet driver associated with a PHY device that a link change event has occured. This routine works by calling the miiMediaUpdate method exported by the parent driver. If the parent device has no miiMediaUpdate method, this routine fails.

RETURNS

OK media update notification succeeds, or **ERROR** otherwise

ERRNO

N/A

SEE ALSO

miiBus

miiBusModeGet( )

NAME

miiBusModeGet( ) – get the current media mode and link status

SYNOPSIS

```
STATUS miiBusModeGet
    (
    VXB_DEVICE_ID pDev,
    UINT32 *      mode,
    UINT32 *      sts
    )
```

DESCRIPTION

This function queries the PHY currently active on the MII bus, specified by pDev, to determine the current link mode and state. The mode and link state are specified using ifmedia definitions (specified in **if_media.h**). These values can be passed directly to callers of the EIOCGIFMEDIA ioctl in drivers that implement ifmedia support.

If no active PHY is currently selected, this routine will fail. If a PHY has been selected (by a call to **miiBusModeSet( )**), its miiModeGet method will be called to query its current setting. Note that this will result in a read of several of the PHY's registers being accessed, including the status register.

RETURNS

OK if reading the current mode/state succeeds, or **ERROR** otherwise

ERRNO
N/A

SEE ALSO
miiBus

miiBusModeSet( )

NAME
miiBusModeSet( ) – set the current media mode

SYNOPSIS
```
STATUS miiBusModeSet
    (
    VXB_DEVICE_ID pDev,
    UINT32        mode
    )
```

DESCRIPTION
This function sets the desired link mode of the MII bus. A bus could potentially have more than one PHY attached, though only one PHY should support any given mode (i.e. you can have one PHY supporting 100baseTX and another supporting 100baseFX, but you can't have two that both support 100baseTX -- there wouldn't be any point).

This routine will search the bus media list for the desired media and make the PHY that supports it the active PHY. The PHY's miiModeSet method will then be invoked to program the PHY for the desired mode. If the desired mode doesn't exist in the list, or if the PHY doesn't export an miiModeSet method, this routine fails.

This routine is typically used by ethernet drivers with ifmedia support to service the EIOCSIFMEDIA ioctl.

RETURNS
OK if setting the mode succeeds, or **ERROR** otherwise

ERRNO
N/A

SEE ALSO
miiBus

miiBusRead( )

NAME
miiBusRead( ) – read a PHY register

SYNOPSIS
```
STATUS miiBusRead
    (
    VXB_DEVICE_ID pDev,
    int           phyAddr,
    int           phyReg,
    UINT16        *regVal
    )
```

DESCRIPTION This function reads a register from a PHY, specified by pDev, at a given address. This routine works by invoking the miiRead method exported by the parent bridge device to which the bus is attached (e.g. the mottsec driver). If the parent bridge does not export an miiRead method, this routine will fail.

RETURNS **OK** read succeeds, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **miiBus**

miiBusRegister( )

NAME **miiBusRegister( )** – register with the vxBus subsystem

SYNOPSIS

```
void miiBusRegister(void)
```

DESCRIPTION This routine registers the miiBus driver with vxBus as a child of all applicable parent bus types, and registers the **VXB_BUSID_MII** bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **miiBus**

miiBusWrite( )

NAME **miiBusWrite( )** – write value to a PHY register

SYNOPSIS

```
STATUS miiBusWrite
    (
    VXB_DEVICE_ID pDev,
    int           phyAddr,
    int           phyReg,
    UINT16         regVal
    )
```

DESCRIPTION This function writes a value to a register from a PHY, specified by pDev, at a given address. This routine works by invoking the miiWrite method exported by the parent bridge device to which the bus is attached (e.g. the mottsec driver). If the parent bridge does not export an miiWrite method, this routine will fail.

RETURNS **OK** write succeeds, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **miiBus**

mkdir( )

NAME **mkdir( )** – make a directory

SYNOPSIS

```
STATUS mkdir
       (
       const char *    dirName         /* directory name */
       )
```

DESCRIPTION This command creates a new directory in a hierarchical file system. The *dirName* string specifies the name to be used for the new directory, and can be either a full or relative pathname.

This call is supported by the VxWorks NFS and dosFs file systems.

RETURNS **OK**, or **ERROR** if the directory cannot be created.

ERRNO Not Available

SEE ALSO **usrFsLib**, **rmdir( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

mlock( )

NAME **mlock( )** – lock specified pages into memory (POSIX)

SYNOPSIS

```
int mlock
```

```
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION This routine guarantees that the specified pages are memory resident. In VxWorks, the *addr* and *len* arguments are ignored, since all pages are memory resident.

RETURNS 0 (**OK**) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

mlockall( )

NAME **mlockall( )** – lock all pages used by a process into memory (POSIX)

SYNOPSIS

```
int mlockall
    (
    int flags
    )
```

DESCRIPTION This routine guarantees that all pages used by a process are memory resident. In VxWorks, the *flags* argument is ignored, since all pages are memory resident.

RETURNS 0 (**OK**) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

mmapShow( )

NAME **mmapShow( )** – show information about memory mapped objects in the system

SYNOPSIS

```
STATUS mmapShow
    (
    char * name
    )
```

DESCRIPTION This routine displays information about objects mapped in the memory space of processes running in the system. Two types of objects are supported: regular files of supported file systems, and shared memory objects. These objects are mapped in the address space of a process with **mmap( )**.

This routine can be used in two ways. With the summary mode, when *name* is **NULL**, this routine lists objects that are mapped in at least one process at the time of the call. Mappings of objects that have been unlinked are shown using this mode.

With the detailed mode, when a file or shared memory object name is specified, this routine lists all memory mappings of the object. For shared memory objects this also includes shared mappings that have been unmapped from all processes by the time this routine is called. Object names that have been unlinked are not accepted.

This routine should be used for debugging purposes only.

EXAMPLE The following example shows the output of summary **mmapShow( )** using the shell's C-interpreter:

```
-> mmapShow

 OBJECT                               UNLINKED  RTP ID     RTP NAME
------------------------------------ -------- ----------
--------------------
/pxTestFs/mmapFile1                  no       0x61746228 <
in/tmMmanFdLib.vxe
                                              0x606d6010 <

in/tmMmanFdLib.vxe
/pxTestFs/mmapFile2                  yes      0x61746228 <
in/tmMmanFdLib.vxe
                                              0x606d6010 <

in/tmMmanFdLib.vxe
value = 0 = 0x0
```

The following example shows the output of summary **mmapShow( )** using the shell's C-interpreter:

```
-> mmapShow "/pxTestFs/mmapFile1"

 ADDRESS    LENGTH     PROT  FLAGS    OFFSET             RTP ID
---------- ---------- ---- -------- ------------------ ----------
0x6306a000 0x00002000  R--   SHARED 0x0000000000000000 0x61746228
0x630d4000 0x00002000  RW-  PRIVATE 0x0000000000000000 0x606d6010
value = 0 = 0x0
```

For the command-interpreter shell, use the **mmap list** command.

RETURNS N/A

ERRNO Not Available

SEE ALSO **mmanShow**

mmuPhysToVirt( )

NAME **mmuPhysToVirt( )** – translate a physical address to a virtual address (ARM)

SYNOPSIS

```
VIRT_ADDR mmuPhysToVirt
    (
    PHYS_ADDR physAddr  /* physical address to be translated */
    )
```

DESCRIPTION This function converts a physical address to a virtual address using the information contained within the sysPhysMemDesc structure of the BSP. This routine may be used both by the BSP MMU initialization and by the vm(Base)Lib code.

If the BSP has a default mapping where physical and virtual addresses are not identical, then it must provide routines to the cache and MMU architecture code to convert between physical and virtual addresses. If the mapping described within the sysPhysMemDesc structure is accurate, then the BSP may use this routine. If it is not accurate, then routines must be provided within the BSP that are accurate.

NOTE This routine simply performs a linear search through the sysPhysMemDesc structure looking for the first entry with an address range that includes the given address. Typically, the performance of this should not be a problem, as this routine will generally be called to translate RAM addresses, and by convention, the RAM entries come first in the structure. If this becomes an issue, the routine could be changed so that a separate structure to sysPhysMemDesc is used, containing the information in a more quickly accessible form. In any case, if this is not satisfactory, the BSP can provide its own routines.

RETURNS the virtual address

ERRNO Not Available

SEE ALSO **mmuMapLib**, mmuVirtToPhys

mmuPro32LibInit( )

NAME **mmuPro32LibInit( )** – initialize module

SYNOPSIS

```
STATUS mmuPro32LibInit
```

```
    (
    int pageSize  /* system pageSize (must be 4096 for i86) */
    )
```

DESCRIPTION Build a dummy translation table that will hold the page table entries for the global translation table. The mmu remains disabled upon completion.

Supervisor Mode Only Not callable from ISR

RETURNS **OK** if no error, **ERROR** otherwise

ERRNO **S_mmuLib_INVALID_PAGE_SIZE**

SEE ALSO **mmuPro32Lib**

mmuPro32Page0UnMap( )

NAME **mmuPro32Page0UnMap( )** – unmap the page zero for **NULL** pointer detection

SYNOPSIS

```
STATUS mmuPro32Page0UnMap (void)
```

DESCRIPTION This routine unmap the page zero for **NULL** pointer access detection.

Not Callable from user level. Not Callable from ISR.

RETURNS **OK** or **ERROR** if mmuPageUnMap fails

ERRNO Not Available

SEE ALSO **mmuPro32Lib**

mmuPro36LibInit( )

NAME **mmuPro36LibInit( )** – initialize module

SYNOPSIS

```
STATUS mmuPro36LibInit
    (
    int pageSize  /* system pageSize (must be 4KB or 2MB) */
    )
```

DESCRIPTION Build a dummy translation table that will hold the page table entries for the global translation table. The mmu remains disabled upon completion.

RETURNS **OK** if no error, **ERROR** otherwise

ERRNO **S_mmuLib_INVALID_PAGE_SIZE**

SEE ALSO **mmuPro36Lib**

mmuPro36Page0UnMap( )

NAME **mmuPro36Page0UnMap( )** – unmap the page zero for **NULL** pointer detection

SYNOPSIS

```
STATUS mmuPro36Page0UnMap (void)
```

DESCRIPTION This routine unmap the page zero for **NULL** pointer access detection.

Not Callable from user level. Not Callable from ISR.

RETURNS **OK** or **ERROR** if mmuPageUnMap fails

ERRNO Not Available

SEE ALSO **mmuPro36Lib**

mmuPro36PageMap( )

NAME **mmuPro36PageMap( )** – map 36bit physical memory page to virtual memory page

SYNOPSIS

```
STATUS mmuPro36PageMap
    (
    MMU_TRANS_TBL * transTbl,        /* translation table */
    VIRT_ADDR       virtualAddress,  /* 32bit virtual address */
    LL_INT          physPage         /* 36bit physical address */
    )
```

DESCRIPTION The 36bit physical page address is entered into the PTE corresponding to the given virtual page. The state of a newly mapped page is undefined.

RETURNS **OK** or **ERROR** if translation table creation failed.

ERRNO Not Available

SEE ALSO **mmuPro36Lib**

mmuPro36Translate( )

NAME **mmuPro36Translate( )** – translate a virtual address to a 36bit physical address

SYNOPSIS

```
STATUS mmuPro36Translate
    (
    MMU_TRANS_TBL * transTbl,      /* translation table */
    VIRT_ADDR       virtAddress,  /* 32bit virtual address */
    LL_INT *        physAddress   /* place to return 36bit result */
    )
```

DESCRIPTION Traverse the translation table and extract the 36bit physical address for the given virtual address from the PTE corresponding to the virtual address.

RETURNS **OK** or **ERROR** if no PTE for given virtual address.

ERRNO Not Available

SEE ALSO **mmuPro36Lib**

mmuShLibInit( )

NAME **mmuShLibInit( )** – Initialize the SH MMU library.

SYNOPSIS

```
STATUS mmuShLibInit
    (
    int pageSize  /* minimum vm page size */
    )
```

DESCRIPTION This routine performs the necessary initialization for the SH MMU library. Initialization consists mainly of initializing processing variables, setting up the processing variables for the AIM and **vmLib** and calling the AIM init function.

RETURNS **OK** or **ERROR** if unsuccessful.

ERRNO

SEE ALSO **mmuShLib**

mmuVirtToPhys( )

NAME **mmuVirtToPhys( )** – translate a virtual address to a physical address (ARM)

SYNOPSIS

```
PHYS_ADDR mmuVirtToPhys
    (
    VIRT_ADDR virtAddr  /* virtual address to be translated */
    )
```

DESCRIPTION This function converts a virtual address to a physical address using the information contained within the sysPhysMemDesc structure of the BSP. This routine may be used both by the BSP MMU initialization and by the vm(Base)Lib code.

If the BSP has a default mapping where physical and virtual addresses are not identical, then it must provide routines to the cache and MMU architecture code to convert between physical and virtual addresses. If the mapping described within the sysPhysMemDesc structure is accurate, then the BSP may use this routine. If it is not accurate, then routines must be provided within the BSP that are accurate.

NOTE This routine simply performs a linear search through the sysPhysMemDesc structure looking for the first entry with an address range that includes the given address. Typically, the performance of this should not be a problem, as this routine will generally be called to translate RAM addresses, and by convention, the RAM entries come first in the structure. If this becomes an issue, the routine could be changed so that a separate structure to sysPhysMemDesc is used, containing the information in a more quickly accessible form. In any case, if this is not satisfactory, the BSP can provide its own routines.

RETURNS the physical address

ERRNO Not Available

SEE ALSO **mmuMapLib**, mmuPhysToVirt

moduleCheck( )

NAME

moduleCheck( ) – verify checksums on all modules loaded in the system

SYNOPSIS

```
STATUS moduleCheck
    (
    int options   /* validation options */
    )
```

DESCRIPTION

This routine verifies the checksums on the sections of all loaded modules. The checksums are compared to original checksums computed when the modules were initialy loaded. If any of the checksums are incorrect, a message is printed to the console, and the routine returns **ERROR**.

By default, only the text section checksums are validated.

Bits in the *options* parameter may be set to control specific checks:

MODCHECK_TEXT
Validate the checksum for the TEXT sections (default).

MODCHECK_DATA
Validate the checksum for the DATA sections.

MODCHECK_BSS
Validate the checksum for the BSS sections.

MODCHECK_RODATA
Validate the checksum for the RODATA sections.

MODCHECK_ALL
Validate the checksum for the all sections.

MODCHECK_NOPRINT
Do not print a message (**moduleCheck( )** still returns **ERROR** on failure.)

See the definitions in **moduleLib.h**

CAVEAT

This routine is a not able to check the integrity of a module at the time of its load. It can only detect corruption subsequent to the load.

RETURNS

OK, or **ERROR** if a checksum is invalid for any module.

ERRNO

Not Available

SEE ALSO

moduleLib

moduleCreate( )

NAME

moduleCreate( ) – create and initialize a module

SYNOPSIS

```
MODULE_ID moduleCreate
    (
    char * name,     /* module name */
    int    format,   /* object module format */
    int    flags     /* <options> passed to loader (see loadModuleAt()) */
    )
```

DESCRIPTION

This routine creates a code module descriptor.

The arguments specify the name of the object module file, the object module format and an argument specifying which symbols to add to the symbol table. See the **loadModuleAt( )** description of *options* for possibles *flags* values.

Space for the new code module descriptor is allocated dynamically.

This function is not intended to be used by code outside of the VxWorks kernel libraries. Documentation is provided for reference only.

RETURNS

The **MODULE_ID** of the newly created module or **NULL** if there is an error.

ERRNO

Not Available

SEE ALSO

loadModuleAt( ), **moduleLib**

moduleCreateHookAdd( )

NAME

moduleCreateHookAdd( ) – add a routine to be called when a module is added

SYNOPSIS

```
STATUS moduleCreateHookAdd
    (
    FUNCPTR moduleCreateHookRtn  /* routine called when module is added */
    )
```

DESCRIPTION

This routine adds the specified routine to a list of routines to be called each time **moduleCreate( )** is called. The specified routine should be declared as follows:

```
    void moduleCreateHookFunc
        (
        MODULE_ID  moduleId  /* the module ID to act upon */
        )
```

This routine is called after all fields of the module ID have been filled in.

NOTE

Modules do not have information about their object sections or segments when they are created. This information is not available until after the entire load process has finished. Therefore functions used as module create hooks should not use the section or segment information associated with a module.

RETURNS

OK or **ERROR** if there was a problem.

ERRNO

Not Available

SEE ALSO

moduleLib, **moduleCreateHookDelete( )**

moduleCreateHookDelete( )

NAME

moduleCreateHookDelete( ) – delete a previously added module create hook routine

SYNOPSIS

```
STATUS moduleCreateHookDelete
    (
    FUNCPTR moduleCreateHookRtn  /* routine called when module is added */
    )
```

DESCRIPTION

This routine removes a specified routine from the list of routines to be called at each **moduleCreate( )** call.

RETURNS

OK, or **ERROR** if the routine is not in the table of module creation hook routines.

ERRNO

Possible errnos set by this routine include:

+ **S_moduleLib_HOOK_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO

moduleLib, **moduleCreateHookAdd( )**

moduleDelete( )

NAME

moduleDelete( ) – delete module ID information

SYNOPSIS

```
STATUS moduleDelete
    (
    MODULE_ID moduleId  /* module to delete */
    )
```

DESCRIPTION

This routine deletes a module descriptor, freeing any space that was allocated for the use of the module ID.

This routine does not free space allocated for the object module itself -- this is done by the unload routine (**unld( )** or **unldByModuleId( )**).

This function is not intended to be used by code outside of the VxWorks kernel libraries. Documentation is provided for reference only.

RETURNS

OK or **ERROR** if there was a problem.

ERRNO

Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO

moduleLib, **unldByModuleId( )**

moduleFindByGroup( )

NAME

moduleFindByGroup( ) – find a module by group number

SYNOPSIS

```
MODULE_ID moduleFindByGroup
    (
    int groupNumber  /* group number to find */
    )
```

DESCRIPTION

This routine searches for a module with a group number matching *groupNumber*.

RETURNS

A **MODULE_ID** corresponding to the first module whose group number matches, or **NULL** if no match is found.

ERRNO

Not Available

SEE ALSO

moduleLib, **moduleIdFigure( )**

moduleFindByName( )

NAME

moduleFindByName( ) – find a module by name

SYNOPSIS

```
MODULE_ID moduleFindByName
    (
    char * moduleName  /* name of module to find */
    )
```

DESCRIPTION

This routine searches for a module with a name matching *moduleName*. The name is the one that was used when the module was loaded.

RETURNS

A **MODULE_ID** corresponding to the module name, or **NULL** if no match is found.

ERRNO

Not Available

SEE ALSO

moduleLib, **moduleFindByNameAndPath( )**

moduleFindByNameAndPath( )

NAME

moduleFindByNameAndPath( ) – find a module by filename and path

SYNOPSIS

```
MODULE_ID moduleFindByNameAndPath
    (
    char * moduleName,  /* file name to find */
    char * pathName     /* path name to find */
    )
```

DESCRIPTION

This routine searches for a module with a name matching *moduleName* and path matching *pathName*. The name and path correspond to the parameters that were passed to the load routine *when the module was loaded*.

EXAMPLES

If the module was loaded using the following name and path:

```
fd = open ("path/to/the/module/to/load/moduleName", O_RDONLY);
moduleLoad (fd, LOAD_GLOBAL_SYMBOLS);
```

then the call to **moduleFindByNameAndPath( )** would be done as:

```
moduleFindByNameAndPath("moduleName", "path/to/the/module/to/load");
```

The path field should be left empty if the module was loaded without any path specified:

```
fd = open ("moduleName", O_RDONLY);
moduleLoad (fd, LOAD_GLOBAL_SYMBOLS);
moduleFindByNameAndPath("moduleName", "");
```

RETURNS A **MODULE_ID**, or **NULL** if no match is found.

ERRNO Not Available

SEE ALSO **moduleLib**

moduleFlagsGet( )

NAME **moduleFlagsGet( )** – get the flags associated with a module ID

SYNOPSIS

```
int moduleFlagsGet
    (
    MODULE_ID moduleId
    )
```

DESCRIPTION This routine returns the flags associated with a module ID. A module's flags correspond to the options passed to the loader when loading the module. See **loadModuleAt( )** reference entry for more information concerning loader flags.

RETURNS The flags associated with the module ID, or zero if the module ID is invalid.

ERRNO Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO **loadModuleAt( )**, **moduleLib**

moduleIdListGet( )

NAME **moduleIdListGet( )** – get a list of loaded modules

SYNOPSIS

```
int moduleIdListGet
    (
    MODULE_ID * idList,     /* Array of module IDs to be filled in  */
    int         maxModules  /* Max modules <idList> can accommodate */
    )
```

DESCRIPTION This routine provides the calling task with a list of all loaded object modules. An unsorted list of module IDs for no more than *maxModules* modules is put into *idList*.

RETURNS The number of modules put into *idList*.

ERRNO Not Available

SEE ALSO **moduleLib**

moduleInfoGet( )

NAME **moduleInfoGet( )** – get information about an object module

SYNOPSIS

```
STATUS moduleInfoGet
    (
    MODULE_ID     moduleId,    /* module to return information about */
    MODULE_INFO * pModuleInfo  /* pointer to module info struct */
    )
```

DESCRIPTION This routine fills in a **MODULE_INFO** structure with information about the specified module. Note that the name field of the **MODULE_INFO** structure is a fixed length (**NAME_MAX**, see *vxParams.h* for actual value); the name of the module will be truncated to fit in the field, if necessary.

RETURNS **OK**, or **ERROR** if the module ID is invalid.

ERRNO Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO **moduleLib**

moduleNameGet( )

NAME **moduleNameGet( )** – get the name associated with a module ID

SYNOPSIS

```
char * moduleNameGet
    (
    MODULE_ID moduleId
    )
```

DESCRIPTION This routine returns a pointer to the name associated with a module ID. Note: this is a pointer to the module library's copy of the name string, so the memory it points to should not be modified.

RETURNS A pointer to the module name corresponding to the module ID, or **NULL** if the module ID is invalid.

ERRNO Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO **moduleLib**

moduleSegFirst( )

NAME **moduleSegFirst( )** – find the first segment in a module

SYNOPSIS

```
SEGMENT_ID moduleSegFirst
    (
    MODULE_ID moduleId  /* module to get first segment of */
    )
```

DESCRIPTION This routine returns the ID of the first segment of a module descriptor.

RETURNS A pointer to the segment ID, or **NULL** if the segment list is empty or the module ID is invalid.

ERRNO Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO **moduleLib**, **moduleSegGet( )**

moduleSegGet( )

NAME **moduleSegGet( )** – get (delete and return) the first segment from a module

SYNOPSIS

```
SEGMENT_ID moduleSegGet
```

```
    (
    MODULE_ID moduleId  /* module to get segment from */
    )
```

DESCRIPTION

This routine returns the ID of the first segment of a module descriptor, and then removes the segment descriptor from the module's segment list.

The memory associated with the segment descriptor is not freed.

This function is not intended to be used by code outside of the VxWorks kernel libraries. Documentation is provided for reference only. Use the routines **moduleSegFirst( )** and **moduleSegNext( )** to retrieve information about a module's segments.

RETURNS

A pointer to the segment ID, or **NULL** if the segment list is empty.

ERRNO

Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO

moduleLib, **moduleSegFirst( )**, **moduleSegNext( )**

moduleSegNext( )

NAME

moduleSegNext( ) – find the next segment in a module

SYNOPSIS

```
SEGMENT_ID moduleSegNext
    (
    SEGMENT_ID segmentId  /* segment whose successor is to be found */
    )
```

DESCRIPTION

This routine returns the ID of the segment in the list immediately following *segmentId*.

RETURNS

A **SEGMENT_ID**, or **NULL** if there is no next segment.

ERRNO

Not Available

SEE ALSO

moduleLib, moduleSegFirst ()

moduleShow( )

NAME

moduleShow( ) – show information about loaded modules

SYNOPSIS

```
STATUS moduleShow
    (
    char * moduleNameOrId,  /* name or ID of the module to show */
    int    options          /* display options */
    )
```

DESCRIPTION

This routine displays information about currently loaded modules and where they are placed in memory. Different information can be obtained depending on the value of the *moduleNameOrId* parameter :

NULL
A summary list of all loaded modules will be displayed. For each module are displayed its base name, ID, group number and the start addresses of the text, data and BSS segments.

A module ID or the name of a loaded module
More information about this specific module will be displayed (namely the sizes of the text, data and BSS segments and the total size of the module).

If the *options* parameter is set to **MODDISPLAY_LONG**, module names longer than 15 characters are displayed on their own line (they would otherwise be truncated). In this case, if **moduleShow( )** is called with a module ID as an argument, the full module path is also printed.

EXAMPLES

Show the list of all modules loaded (C shell):

```
-> moduleShow ()

MODULE NAME     MODULE ID  GROUP #    TEXT START DATA START  BSS START
--------------- ---------- ---------- ---------- ---------- ----------
versionDotOWith 0x616fc520          2 0x60532000 0x60534000 NO SEGMENT
versionDotO15.o 0x616fbe98          3 0x60536000 0x60538000 NO SEGMENT
ctdt.o          0x60534020          4 NO SEGMENT 0x61700000 NO SEGMENT
value = 0 = 0x0
```

Display information about a particular module (C shell):

```
-> moduleShow (0x616fc520, 0)

MODULE NAME     MODULE ID  GROUP #    TEXT START DATA START  BSS START
--------------- ---------- ---------- ---------- ---------- ----------
versionDotOWith 0x616fc520          2 0x60532000 0x60534000 NO SEGMENT

Size of text segment:         58
Size of data segment:         16
Size of bss  segment:          0
Total size          :         74
value = 0 = 0x0
```

Display full names for all modules loaded (C shell):

```
-> moduleShow (0, 1)

MODULE NAME     MODULE ID  GROUP #    TEXT START DATA START  BSS START
--------------- ---------- ---------- ---------- ---------- ----------
versionDotOWithALongName.o
                0x616fc520          2 0x60532000 0x60534000 NO SEGMENT
versionDotO15.o 0x616fbe98          3 0x60536000 0x60538000 NO SEGMENT
ctdt.o          0x60534020          4 NO SEGMENT 0x61700000 NO SEGMENT
value = 0 = 0x0
```

Display full name and path for a particular module (C shell):

```
-> moduleShow (0x616fc520, 1)

MODULE NAME     MODULE ID  GROUP #    TEXT START DATA START  BSS START
--------------- ---------- ---------- ---------- ---------- ----------
versionDotOWithALongName.o
                0x616fc520          2 0x60532000 0x60534000 NO SEGMENT

Size of text segment:         58
Size of data segment:         16
Size of bss  segment:          0
Total size          :         74

Module path:
huelgoat:/folk/joe/target/proj/linux_gnu/default
value = 0 = 0x0
```

It is also possible to pass a module name instead of a module ID to the **moduleShow( )** command. Thus :

```
-> moduleShow ("versionDotOWithALongName.o", 1)
```

would give the same output as above.

RETURNS **OK**, or **ERROR** if the module is not found.

ERRNO Not Available

SEE ALSO **moduleLib**, *VxWorks Kernel Programmer's Guide: `Target Shell`*, *Workbench User's Guide: `Wind Shell`*

mountdInit( )

NAME **mountdInit( )** – initialize the mount daemon

SYNOPSIS

```
STATUS mountdInit
```

```
    (
    int      priority,   /* priority of the mount daemon             */
    int      stackSize,  /* stack size of the mount daemon           */
    FUNCPTR authHook,    /* hook to run to authorize each request    */
    int      nExports,   /* maximum number of exported file systems */
    int      options     /* Currently unused                         */
    )
```

DESCRIPTION This routine spawns a mount daemon if one does not already exist. Defaults for the *priority* and *stackSize* arguments are in the global variables **mountdPriorityDefault** and **mountdStackSizeDefault**, and are initially set to **MOUNTD_PRIORITY_DEFAULT** and **MOUNTD_STACKSIZE_DEFAULT** respectively.

Normally, no authorization checking is performed by either mountd or nfsd. To add authorization checking, set *authHook* to point to a routine declared as follows:

```
nfsstat routine
    (
    int                progNum,      /* RPC program number */
    int                versNum,      /* RPC program version number */
    int                procNum,      /* RPC procedure number */
    struct sockaddr_in clientAddr,   /* address of the client */
    void             *  mountdArg    /* argument of the call */
    )
```

The mountdArg will be of type MOUNT3D_ARGUMENT when versNum is 3 and it will be of type **MOUNTD_ARGUMENT** when versNum is 1. The user routine must typecast the mountdArg accoringly and use it. The definitions of MOUNT3D_ARGUMENT & **MOUNTD_ARGUMENT** are available in **mountd.h** file.

The *authHook* callback must return **OK** if the request is authorized, and any defined NFS error code (usually **NFSERR_ACCESS**) if not.

RETURNS **OK**, or **ERROR** if the mount daemon could not be correctly initialized.

ERRNO Not Available

SEE ALSO **mountd**

mqPxDescObjIdGet( )

NAME **mqPxDescObjIdGet( )** – returns the **OBJ_ID** associated with a mqd_t descriptor

SYNOPSIS

```
OBJ_ID mqPxDescObjIdGet
    (
    mqd_t mqdes  /* message queue descriptor */
    )
```

DESCRIPTION The message queue object identifier (**OBJ_ID**) is returned given a POSIX message queue descriptor (mqd_t).

RETURNS **OBJ_ID**, or **NULL** if the message queue descriptor is invalid.

ERRNO None

SEE ALSO **mqPxLib**

mqPxLibInit( )

NAME **mqPxLibInit( )** – initialize the POSIX message queue library

SYNOPSIS

```
int mqPxLibInit
    (
    int hashSize  /* not used */
    )
```

DESCRIPTION This routine initializes the POSIX message queue facility.

RETURNS **OK** or **ERROR**.

ERRNO N/A

SEE ALSO **mqPxLib**

mqPxShow( )

NAME **mqPxShow( )** – display message queue internals

SYNOPSIS

```
STATUS mqPxShow
    (
    mqd_t mqDesc,
    int   level
    )
```

DESCRIPTION This routine displays information on a POSIX message queue.

RETURNS **OK** or **ERROR** if the descriptor is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**
message queue is invalid.

SEE ALSO **mqPxShow**

mqPxShowInit( )

NAME **mqPxShowInit( )** – initialize the POSIX message queue show facility

SYNOPSIS

```
STATUS mqPxShowInit (void)
```

DESCRIPTION This routine links the POSIX message queue show routine 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**.

If you use the project facility, select **INCLUDE_POSIX_MQ_SHOW**.

RETURNS **OK** always

ERRNO N/A

SEE ALSO **mqPxShow**

mq_close( )

NAME **mq_close( )** – close a message queue (POSIX)

SYNOPSIS

```
int mq_close
    (
    mqd_t mqdes  /* message queue descriptor */
    )
```

DESCRIPTION This routine is used to indicate that the calling task is finished with the specified message queue *mqdes*. The **mq_close( )** call deallocates any system resources allocated by the system for use by this task for its message queue. The behavior of a task that is blocked on either a **mq_send( )** or **mq_receive( )** is undefined when **mq_close( )** is called. The *mqdes* parameter will no longer be a valid message queue ID.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**) if the message queue is closed successfully, otherwise -1 (**ERROR**).

ERRNO

EBADF
The *mqdes* argument is not a valid message queue descriptor.

SEE ALSO

mqPxLib, **mq_open( )**

mq_getattr( )

NAME

mq_getattr( ) – get message queue attributes (POSIX)

SYNOPSIS

```
int mq_getattr
    (
    mqd_t            mqdes,   /* message queue descriptor */
    struct mq_attr * pMqStat  /* buffer in which to return attributes */
    )
```

DESCRIPTION

This routine gets status information and attributes associated with a specified message queue *mqdes*. Upon return, the following members of the **mq_attr** structure referenced by *pMqStat* will contain the values set when the message queue was opened but with modifications made by subsequent calls to **mq_setattr( )**:

q_flags
May be modified by **mq_setattr( )**.

The following members were set at message queue creation:

mq_maxmsg
Maximum number of messages.

mq_msgsize
Maximum message size.

The following member contains the current state of the message queue.

mq_curmsgs
The number of messages currently in the queue.

RETURNS

0 (**OK**) if message attributes can be determined, otherwise -1 (**ERROR**).

ERRNO

EBADF
The *mqes* argument is not a valid message queue descriptor.

SEE ALSO

mqPxLib, **mq_open( )**, **mq_send( )**, **mq_setattr( )**

mq_notify( )

NAME

mq_notify( ) – notify a task that a message is available on a queue (POSIX)

SYNOPSIS

```
int mq_notify
    (
    mqd_t                    mqdes,         /* message queue descriptor */
    const struct sigevent * pNotification  /* real-time signal */
    )
```

DESCRIPTION

If *pNotification* is not **NULL**, this routine attaches the specified *pNotification* request by the calling task to the specified message queue *mqdes* associated with the calling task. The real-time signal specified by *pNotification* will be sent to the task when the message queue changes from empty to non-empty. If a task has already attached a notification request to the message queue, all subsequent attempts to attach a notification to the message queue will fail. A task can get notifications from multiple messages queues.

If *pNotification* is **NULL** and the task has previously attached a notification request to the message queue, the attached notification request is detached and the queue is available for another task to attach a notification request.

If a notification request is attached to a message queue and any task is blocked in **mq_receive( )** waiting to receive a message when a message arrives at the queue, then the appropriate **mq_receive( )** will be completed and the notification request remains pending.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**) if successful, otherwise -1 (**ERROR**).

ERRNO

EBADF
The *mqes* argument is not a valid message queue descriptor.

EBUSY
A task is already registered for notification by the message queue.

EINVAL
This task is trying to remove the registration of another task.

SEE ALSO **mqPxLib**, **mq_open( )**, **mq_send( )**

mq_open( )

NAME **mq_open( )** – open a message queue (POSIX)

SYNOPSIS

```
mqd_t mq_open
    (
    const char * mqName,  /* name of queue to open */
    int          oflags,  /* open flags */
    ...                   /* extra optional parameters */
    )
```

DESCRIPTION This routine establishes a connection between a named message queue and the calling task. After a call to **mq_open( )**, the task can reference the message queue using the address returned by the call. The message queue remains usable until the queue is closed by a successful call to **mq_close( )**.

The message queue must have a name. **NULL** and empty strings result in **EINVAL**. If the *name* begins with the slash character, then it is treated as a public message queue. All RTPs can open their own references to the public message queue by using its name. If the *name* does not begin with the slash character, then it is treated as a private message queue and RTPs cannot get access to it.

The following flag bits can be set in *oflags*:

O_RDONLY
Open the message queue for receiving messages. The task can use the returned message queue descriptor with **mq_receive( )**, but not **mq_send( )**.

O_WRONLY
Open the message queue for sending messages. The task can use the returned message queue descriptor with **mq_send( )**, but not **mq_receive( )**.

O_RDWR
Open the queue for both receiving and sending messages. The task can use any of the functions allowed for **O_RDONLY** and **O_WRONLY**.

Any combination of the following flags can be specified in *oflags*. These control whether the message queue is created or merely accessed by the **mq_open( )** call.

O_CREAT
This flag is used to create a message queue if it does not already exist. If **O_CREAT** is set and the message queue already exists, then **O_CREAT** has no effect except as noted below under **O_EXCL**. Otherwise, **mq_open( )** creates a message queue. The **O_CREAT** flag requires a third and fourth argument: *mode*, which is of type **mode_t**, and *pAttr*, which is of type pointer to an **mq_attr** structure. The value of *mode* has no effect in this

implementation. If *pAttr* is **NULL**, the message queue is created with a **MQ_NUM_MSG_DEFAULT** messages of size **MQ_MSG_SIZE_DEFAULT**. If *pAttr* is non-**NULL**, the message queue attributes **mq_maxmsg** and **mq_msgsize** are set to the values of the corresponding members in the **mq_attr** structure referred to by *pAttr*; if either attribute is less than or equal to zero, an error is returned and errno is set to **EINVAL**.

O_EXCL
This flag is used to test whether a message queue already exists. If **O_EXCL** and **O_CREAT** are set, **mq_open( )** fails if the message queue name exists.

O_NONBLOCK
The setting of this flag is associated with the open message queue descriptor. If this flag is set, then the **mq_send( )** and **mq_receive( )** do not wait for resources or messages that are not currently available. Instead, they fail with errno set to **EAGAIN**.

The **mq_open( )** call does not add or remove messages from the queue.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

NOTE

Some POSIX functionality is not yet supported:

- A message queue cannot be closed with calls to **_exit( )** or **exec( )**.
- A message queue cannot be implemented as a file.
- Message queue names will not appear in the file system.

RETURNS

A message queue descriptor, otherwise -1 (**ERROR**).

ERRNO

EEXIST
O_CREAT and **O_EXCL** are set and the message queue already exists.

ENOENT
O_CREAT is not set and the message queue does not exist.

ENOSPC
There is insufficient space for the creation of the new message queue.

EINVAL

- The specified *name* is invalid.
- An invalid combination of *oflags* is specified.
- **O_CREAT** is specified in *oflags*, the value of *pAttr* is not **NULL** and either *mq_maxmsg* or *mq_msgsize* is less than or equal to zero.

SEE ALSO **mqPxLib**, **mq_send( )**, **mq_receive( )**, **mq_close( )**, **mq_setattr( )**, **mq_getattr( )**, **mq_unlink( )**

mq_receive( )

NAME

mq_receive( ) – receive a message from a message queue (POSIX)

SYNOPSIS

```
ssize_t mq_receive
    (
    mqd_t     mqdes,    /* message queue descriptor */
    void    * pMsg,     /* buffer to receive message */
    size_t    msgLen,   /* size of buffer, in bytes */
    int     * pMsgPrio  /* if not NULL, priority of message */
    )
```

DESCRIPTION

This routine receives the oldest of the highest priority message from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msgLen* argument, is less than the **mq_msgsize** attribute of the message queue, **mq_receive( )** will fail and return an error. Otherwise, the selected message is removed from the queue and copied to *pMsg*.

If *pMsgPrio* is not **NULL**, the priority of the selected message will be stored in *pMsgPrio*.

If the message queue is empty and **O_NONBLOCK** is not set in the message queue's description associated with *mqdes*, **mq_receive( )** will block until a message is added to the message queue, or until it is interrupted by a signal. If more than one task is waiting to receive a message when a message arrives at an empty queue, the task of highest priority will be selected to receive the message. If the specified message queue is empty and **O_NONBLOCK** is set in the message queue's description associated with *mqdes*, no message is removed from the queue, and **mq_receive( )** returns an error.

The non-negative size value of the msgLen is not limited, and if a negative value is specified for msgLen, the negativitiy of that value will be ignored.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The length of the selected message in bytes, otherwise -1 (**ERROR**).

ERRNO

EAGAIN
O_NONBLOCK was set in the message queue description associated with *mqdes*, and the specified message queue is empty.

EBADF
The *mqdes* argument is not a valid message queue descriptor open for for reading.

EMSGSIZE
The specified message buffer size, *msgLen*, is less than the message size attribute of the message queue.

EINVAL
The *pMsg* pointer is invalid.

EINTR
A signal was received while blocking on the message queue. This error only occurs for an RTP task.

SEE ALSO **mqPxLib**, **mq_send( )**

mq_send( )

NAME **mq_send( )** – send a message to a message queue (POSIX)

SYNOPSIS

```
int mq_send
    (
    mqd_t        mqdes,   /* message queue descriptor */
    const void * pMsg,    /* message to send */
    size_t       msgLen,  /* size of message, in bytes */
    int          msgPrio  /* priority of message */
    )
```

DESCRIPTION This routine adds the message *pMsg* to the message queue *mqdes*. The *msgLen* parameter specifies the length of the message in bytes pointed to by *pMsg*. The value of *pMsg* must be less than or equal to the **mq_msgsize** attribute of the message queue, or **mq_send( )** will fail.

If the message queue is not full, **mq_send( )** will behave as if the message is inserted into the message queue at the position indicated by the *msgPrio* argument. A message with a higher numeric value for *msgPrio* is inserted before messages with a lower value. The value of *msgPrio* must be less than **MQ_PRIO_MAX**.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue, **mq_send( )** will block until space becomes available to queue the message, or until it is interrupted by a signal. If the message queue is full and **O_NONBLOCK** is set in the message queue's descriptions associated with *mqdes*, the message is not queued, and **mq_send( )** returns an error.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. If

mq_send( ) is called from an interrupt service routine, it will behave as if the **O_NONBLOCK** flag were set.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), otherwise -1 (**ERROR**).

ERRNO

EAGAIN
O_NONBLOCK was set in the message queue description associated with *mqdes*, and the specified message queue is full.

EBADF
The *mqdes* argument is not a valid message queue descriptor open for for writing.

EMSGSIZE
The specified message length, *msgLen*, exceeds the message size attribute of the message queue.

EINVAL

- The value of *msgPrio* is greater than or equal to **MQ_PRIO_MAX**.
- The *pMsg* pointer is invalid.

EINTR
The request has been interrupted by a signal.

SEE ALSO

mqPxLib, **mq_receive( )**

mq_setattr( )

NAME

mq_setattr( ) – set message queue attributes (POSIX)

SYNOPSIS

```
int mq_setattr
    (
    mqd_t                    mqdes,       /* message queue descriptor */
    const struct mq_attr * pMqStat,     /* new attributes */
    struct mq_attr *        pOldMqStat  /* old attributes */
    )
```

DESCRIPTION

This routine sets attributes associated with the specified message queue *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure are set to the specified values upon successful completion of the call:

mq_flags
The value of the **O_NONBLOCK** flag.

If *pOldMqStat* is non-**NULL**, **mq_setattr( )** will store, in the location referenced by *pOldMqStat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to **mq_getattr( )** at that point.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS
0 (**OK**) if attributes are set successfully, otherwise -1 (**ERROR**).

ERRNO
EBADF
The *mqes* argument is not a valid message queue descriptor.

SEE ALSO
mqPxLib, **mq_open( )**, **mq_send( )**, **mq_getattr( )**

mq_unlink( )

NAME
mq_unlink( ) – remove a message queue (POSIX)

SYNOPSIS
```
int mq_unlink
    (
    const char * mqName  /* name of message queue */
    )
```

DESCRIPTION
This routine removes the message queue named by the pathname *mqName*. After a successful call to **mq_unlink( )**, a call to **mq_open( )** on the same message queue will fail if the flag **O_CREAT** is not set. If one or more tasks have the message queue open when **mq_unlink( )** is called, removal of the message queue is postponed until all references to the message queue have been closed by **mq_close( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS
0 (**OK**) if the message queue is unlinked successfully, otherwise -1 (**ERROR**).

ERRNO
ENOENT
A message queue with the specified name, *mqName*, does not exist.

SEE ALSO **mqPxLib**, **mq_close( )**, **mq_open( )**

msgQClose( )

NAME **msgQClose( )** – close a named message queue

SYNOPSIS

```
STATUS msgQClose
    (
    MSG_Q_ID msgQId  /* message queue ID to close */
    )
```

DESCRIPTION This routine closes a named message queue and decrements its reference counter. In the case where the counter becomes zero, the message queue is deleted if:

- It has been already removed from the name space by a call to **msgQUnlink( )**.
- It was created with the **OM_DESTROY_ON_LAST_CALL** option.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

S_objLib_OBJ_INVALID_ARGUMENT
The message queue ID is **NULL**.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The message queue is not named.

S_objLib_OBJ_DESTROY_ERROR
An error was detected while deleting the message queue.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **msgQOpen**, **msgQOpen( )**, **msgQUnlink( )**

msgQCreate( )

NAME

msgQCreate( ) – create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,        /* max messages that can be queued */
    int maxMsgLength,   /* max bytes in a message */
    int options         /* message queue options */
    )
```

DESCRIPTION

This routine creates a message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. The routine returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO (0x00)
Queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)
Queue pended tasks in priority order.

MSG_Q_EVENTSEND_ERR_NOTIFY (0x02)
When a message is sent, if a task is registered for events and the actual sending of events fails, a value of **ERROR** is returned and **errno** is set. This option is off by default.

MSG_Q_INTERRUPTIBLE (0x04)
Signal sent to a RTP task pended on a message queue created with this option, would make the task ready and return **ERROR** with errno set to **EINTR**. This option has no affect for a kernel task pended on the same message queue created with this option. This option is off by default.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

MSG_Q_ID, or **NULL** if error

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to create the queue as specified.

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from interrupt level.

S_msgQLib_INVALID_MSG_LENGTH
Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT
Negative maxMsgs specified.

S_msgQLib_INVALID_QUEUE_TYPE
Invalid pending queue type specified.

S_msgQLib_ILLEGAL_OPTIONS
Illegal option bits were specified.

SEE ALSO **msgQLib**, **msgQSmLib**

msgQDelete( )

NAME **msgQDelete( )** – delete a message queue

SYNOPSIS

```
STATUS msgQDelete
    (
    MSG_Q_ID msgQId  /* message queue to delete */
    )
```

DESCRIPTION This routine deletes a message queue. All tasks pending on either **msgQSend( )** or **msgQReceive( )**, or pending for the reception of events meant to be sent from the message queue, unblock and return **ERROR**. When this function returns, *msgQId* is no longer a valid message queue ID.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK** on success or **ERROR** otherwise

ERRNO **S_objLib_OBJ_ID_ERROR**
The message queue ID is invalid.

S_intLib_NOT_ISR_CALLABLE
The routine cannot be called from interrupt level.

S_smObjLib_NO_OBJECT_DESTROY
Deleting a shared message queue is not permitted.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Deleting a named message queue is not permitted.

SEE ALSO **msgQLib**, **msgQSmLib**

msgQEvStart( )

NAME

msgQEvStart( ) – start the event notification process for a message queue

SYNOPSIS

```
STATUS msgQEvStart
    (
    MSG_Q_ID msgQId,  /* msg Q for which to register events */
    UINT32   events,  /* 32 possible events                 */
    UINT8    options  /* event-related msg Q options        */
    )
```

DESCRIPTION

This routine turns on the event notification process for a given message queue, registering the calling task on that queue. When a message arrives on the queue and no receivers are pending, the events specified are sent to the registered task. A task can always overwrite its own registration.

The *events* are user-defined. For more information, see the reference entry for **eventLib**.

The *options* parameter is used for three user options:

- Specify whether the events are to be sent only once or every time a message arrives until **msgQEvStop( )** is called.
- Specify if another task can subsequently register itself while the calling task is still registered. If so specified, the existing task registration will be overwritten without any warning.
- Specify if events are to be sent immediately in the case a message is waiting to be picked up.

Here are the possible values to be used in the option field:

EVENTS_SEND_ONCE (0x1)
The message queue will send the events only once.

EVENTS_ALLOW_OVERWRITE (0x2)
Subsequent registrations from other tasks can overwrite the current one.

EVENTS_SEND_IF_FREE (0x4)
The registration process will send events if a message is present on the message queue when **msgQEvStart( )** is called.

EVENTS_OPTIONS_NONE (0x0)
Must be passed to the *options* parameter if none of the other three options are used.

WARNING

This routine cannot be called from interrupt level.

WARNING

Task preemption can allow a **msgQDelete( )** to be performed between the calls to **msgQEvStart( )** and **eventReceive( )**. This would prevent the task from ever receiving the events wanted from the message queue.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK on success, or **ERROR**

ERRNO

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

S_eventLib_ALREADY_REGISTERED
A task is already registered on the message queue.

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from interrupt level.

S_eventLib_EVENTSEND_FAILED
The user chose to send events immediately and that operation failed.

S_eventLib_ZERO_EVENTS
The user passed in a value of zero to the *events* parameter.

SEE ALSO

msgQEvLib, **eventLib**, **msgQLib**, **msgQEvStop( )**

msgQEvStop( )

NAME

msgQEvStop( ) – stop the event notification process for a message queue

SYNOPSIS

```
STATUS msgQEvStop
    (
    MSG_Q_ID msgQId
    )
```

DESCRIPTION

This routine turns off the event notification process for a given message queue. This allows another task to register itself for event notification on that particular message queue. The routine must be called by the task that is already registered on that particular message queue.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK on success, or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

S_intLib_NOT_ISR_CALLABLE
The routine was called from interrupt level.

S_eventLib_TASK_NOT_REGISTERED
The routine was not called by the registered task.

SEE ALSO

msgQEvLib, **eventLib**, **msgQLib**, **msgQEvStart( )**

msgQInfoGet( )

NAME

msgQInfoGet( ) – get information about a message queue

SYNOPSIS

```
STATUS msgQInfoGet
    (
    MSG_Q_ID     msgQId,  /* message queue to query */
    MSG_Q_INFO * pInfo    /* where to return msg info */
    )
```

DESCRIPTION

This routine gets information about the state and contents of a message queue. The parameter *pInfo* is a pointer to a structure of type **MSG_Q_INFO** defined in **msgQLib.h** as follows:

```
 typedef struct                /* MSG_Q_INFO */
    {
    int     numMsgs;        /* OUT: number of messages queued           */
    int     numTasks;       /* OUT: number of tasks waiting on msg q    */
    int     sendTimeouts;  /* OUT: count of send timeouts              */
    int     recvTimeouts;  /* OUT: count of receive timeouts           */
    int     options;        /* OUT: options with which msg q was created */
    int     maxMsgs;        /* OUT: max messages that can be queued     */
    int     maxMsgLength;  /* OUT: max byte length of each message     */
    int     taskIdListMax; /* IN: max tasks to fill in taskIdList      */
    int *   taskIdList;    /* PTR: array of task IDs waiting on msg q  */
    int     msgListMax;    /* IN: max msgs to fill in msg lists        */
    char ** msgPtrList;    /* PTR: array of msg ptrs queued to msg q   */
    int *   msgLenList;    /* PTR: array of lengths of msgs            */
    } MSG_Q_INFO;
```

If a message queue is empty, there may be tasks blocked on receiving. If a message queue is full, there may be tasks blocked on sending. This can be determined as follows:

- If **numMsgs** is 0, then **numTasks** indicates the number of tasks blocked on receiving.
- If **numMsgs** is equal to **maxMsgs**, then **numTasks** is the number of tasks blocked on sending.

- If **numMsgs** is greater than 0 but less than **maxMsgs**, then **numTasks** is 0.

A list of pointers to the messages queued and their lengths can be obtained by setting **msgPtrList** and **msgLenList** to the addresses of arrays to receive the respective lists, and setting **msgListMax** to the maximum number of elements in those arrays. If either list pointer is **NULL**, no data is returned for that array.

No more than **msgListMax** message pointers and lengths are returned, although **numMsgs** is always returned with the actual number of messages queued.

For example, if the caller supplies a **msgPtrList** and **msgLenList** with room for 10 messages and sets **msgListMax** to 10, but there are 20 messages queued, then the pointers and lengths of the first 10 messages in the queue are returned in **msgPtrList** and **msgLenList**, but **numMsgs** is returned with the value 20.

A list of the task IDs of tasks blocked on the message queue can be obtained by setting **taskIdList** to the address of an array to receive the list, and setting **taskIdListMax** to the maximum number of elements in that array. If **taskIdList** is **NULL**, then no task IDs are returned. No more than **taskIdListMax** task IDs are returned, although **numTasks** is always returned with the actual number of tasks blocked.

For example, if the caller supplies a **taskIdList** with room for 10 task IDs and sets **taskIdListMax** to 10, but there are 20 tasks blocked on the message queue, then the IDs of the first 10 tasks in the blocked queue are returned in **taskIdList**, but **numTasks** is returned with the value 20.

The tasks returned in **taskIdList** may be blocked for either send or receive. As noted above this can be determined by examining **numMsgs**.

The variables **sendTimeouts** and **recvTimeouts** are the counts of the number of times **msgQSend( )** and **msgQReceive( )** respectively returned with a timeout.

The variables **options**, **maxMsgs**, and **maxMsgLength** are the parameters with which the message queue was created.

WARNING

The information returned by this routine is not static and may be obsolete by the time it is examined. In particular, the lists of task IDs or message pointers may no longer be valid. However, the information is obtained atomically; it is an accurate snapshot of the state of the message queue at the time of the call. This information is generally used for debugging purposes only.

WARNING

The current implementation of this routine locks out interrupts while obtaining the information. This can compromise the overall interrupt latency of the system. Generally this routine should be used for debugging purposes only.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK or **ERROR**.

ERRNO

S_smObjLib_NOT_INITIALIZED
The shared memory message queue library (VxMP Option) was not initialized.

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

SEE ALSO

msgQInfo

msgQInitialize( )

NAME

msgQInitialize( ) – initialize a pre-allocated message queue

SYNOPSIS

```
MSG_Q_ID msgQInitialize
    (
    char * pMsgQMem,       /* pointer to msg queue to initialize */
    int    maxMsgs,        /* max messages that can be queued */
    int    maxMsgLength,   /* max bytes in a message */
    int    options         /* message queue options */
    )
```

DESCRIPTION

This routine initializes a pre-allocated message queue structure and message pool memory. The message pool memory must be capable of holding up to *maxMsgs* messages, each of up to *maxMsgLength* bytes long. Parameter *pMsgPool* points to the buffer to be used for holding queued messages. *pMsgPool* must point to a 4 byte aligned buffer whose size is (*maxMsgs* * **MSG_NODE_SIZE** (*maxMsgLength*)).

The queue can be created with the following options:

MSG_Q_FIFO (0x00)
Queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)
Queue pended tasks in priority order.

MSG_Q_EVENTSEND_ERR_NOTIFY (0x02)
When a message is sent, if a task is registered for events and the actual sending of events fails, a value of **ERROR** is returned and **errno** is set. This option is off by default.

MSG_Q_INTERRUPTIBLE (0x04)
A signal sent to an RTP task pended on a message queue created with this option would make the task ready and return **ERROR** with errno set to **EINTR**. This option has no affect for kernel tasks pended on the same message queue created with this option. This option is disabled by default.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

The following example illustrates use of the **VX_MSG_Q** macro and this function together to instantiate a message queue statically (without using any dynamic memory allocation):

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

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

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

RETURNS The **MSG_Q_ID**, or **NULL** on error.

ERRNO N/A

SEE ALSO **msgQLib**, **msgQCreate( )**

msgQNumMsgs( )

NAME **msgQNumMsgs( )** – get the number of messages queued to a message queue

SYNOPSIS

```
int msgQNumMsgs
    (
    FAST MSG_Q_ID msgQId  /* message queue to examine */
    )
```

DESCRIPTION This routine returns the number of messages currently queued to a specified message queue.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS The number of messages queued, or **ERROR**

ERRNO

S_smObjLib_NOT_INITIALIZED
The shared memory message queue library (VxMP Option) was not initialized.

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

SEE ALSO

msgQLib, **msgQSmLib**

msgQOpen( )

NAME

msgQOpen( ) – open a message queue

SYNOPSIS

```
MSG_Q_ID msgQOpen
    (
    const char * name,           /* message queue name */
    int          maxMsgs,        /* max messages that can be queued */
    int          maxMsgLength,   /* max bytes in a message */
    int          options,        /* message queue options */
    int          mode,           /* creation mode */
    void *       context         /* context value */
    )
```

DESCRIPTION

This routine opens a message queue, which means that it searches the name space and returns the **MSG_Q_ID** of an existing message queue with *name*. If none is found, it creates a new one with *name* according on the flags set in the mode parameter.

The argument *name* is mandatory. **NULL** or empty strings are not allowed.

There are two name spaces available in which **msgQOpen( )** can perform the search. The name space searched is dependent upon the first character in the *name* parameter. When this character is a forward slash **/**, the **public** name space is searched; otherwise the **private** name space is searched. Similarly, if a message queue is created, the first character in *name* specifies the name space that contains the message queue.

Message queues created by this routine can not be deleted with **msgQDelete( )**. Instead, a **msgQClose( )** must be issued for every **msgQOpen( )**. Then the message queue is deleted when it is removed from the name space by a call to **msgQUnlink( )**. Alternatively, the message queue can be previously removed from the name space, and deleted during the last **msgQClose( )**.

A description of the *mode* and *context* arguments follows. See the reference entry for **msgQCreate( )** for a description of the remaining arguments.

mode
The mode parameter passed to this routine consists of the opening flags, which can be set using a bitwise-OR:

OM_CREATE
Create a message queue if none is found.

OM_EXCL
When set jointly with the **OM_CREATE** flag, create a new message queue without trying to open an existing one. The call fails if *name* causes a name clash. This flag has no effect if the flag **OM_CREATE** is not set.

OM_DELETE_ON_LAST_CLOSE
Only used when a message queue is created. If set, the message queue is deleted during the last **msgQClose( )** call, independently of whether **msgQUnlink( )** was previously called or not.

context
Context value assigned to the created message queue. This value is not actually used by VxWorks. Instead, the context value can be used by OS extensions to implement object permissions, for example.

Unlike private objects, a public message queue is not automatically reclaimed when an application terminates. Note that nevertheless, a **msgQClose( )** is issued on every application's outstanding **msgQOpen( )**. Therefore, a public message queue can effectively be deleted, if during this process it is closed for the last time, and it is already unlinked or it was created with the **OM_DELETE_ON_LAST_CLOSE** flag.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The **MSG_Q_ID** of the opened message queue, or **NULL** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid.

S_msgQLib_INVALID_MSG_LENGTH
Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT
Negative maxMsgs specified.

S_objLib_OBJ_NOT_FOUND
The **OM_CREATE** flag was not set in the *mode* argument and a message queue matching *name* was not found.

S_objLib_OBJ_NAME_CLASH
The **OM_CREATE** and **OM_EXCL** flags were set and a name clash was detected when creating the message queue.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **msgQOpen**, **msgQUnlink( )**, **msgQClose( )**

msgQOpenInit( )

NAME **msgQOpenInit( )** – initialize the message queue open facility

SYNOPSIS

```
void msgQOpenInit (void)
```

DESCRIPTION This routine links the message queue creation routine with the open facility into the VxWorks system. It is called automatically when the message queue facility is configured into VxWorks by either defining the **INCLUDE_OBJ_OPEN** and **INCLUDING_MSG_Q** components in **config.h** or selecting **INCLUDE_OBJ_OPEN** and **INCLUDING_MSG_Q** in the project facility.

RETURNS N/A

ERRNO N/A

SEE ALSO **msgQOpen**

msgQReceive( )

NAME **msgQReceive( )** – receive a message from a message queue

SYNOPSIS

```
int msgQReceive
    (
    MSG_Q_ID msgQId,     /* message queue from which to receive */
    char *   buffer,     /* buffer to receive message */
    UINT     maxNBytes,  /* length of buffer */
    int      timeout     /* ticks to wait */
    )
```

DESCRIPTION This routine receives a message from the message queue *msgQId*. The received message is copied into the specified *buffer*, which is *maxNBytes* in length. If the message is longer than *maxNBytes*, the remainder of the message is discarded (no error indication is returned).

The *timeout* parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when **msgQReceive( )** is called. The *timeout* parameter can also have the following special values:

NO_WAIT (0)
Return immediately, whether a message has been received or not.

WAIT_FOREVER (-1)
Never time out.

WARNING

This routine must not be called by interrupt service routines.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The number of bytes copied to *buffer*, or **ERROR**

ERRNO

S_smObjLib_NOT_INITIALIZED
The shared memory message queue library (VxMP Option) was not initialized.

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

S_objLib_OBJ_DELETED
The message queue was deleted while the calling task was pended.

S_objLib_OBJ_UNAVAILABLE
No message was available and the **NO_WAIT** timeout was specified.

S_objLib_OBJ_TIMEOUT
A timeout occurred while waiting for a message to become available.

S_msgQLib_INVALID_MSG_LENGTH
The message length exceeds the limit.

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from interrupt level.

SEE ALSO

msgQLib, **msgQSmLib**, **msgQSend( )**

msgQSend( )

NAME

msgQSend( ) – send a message to a message queue

SYNOPSIS

```
STATUS msgQSend
    (
    MSG_Q_ID msgQId,   /* message queue on which to send */
    char *   buffer,   /* message to send */
    UINT     nBytes,   /* length of message */
    int      timeout,  /* ticks to wait */
    int      priority  /* MSG_PRI_NORMAL or MSG_PRI_URGENT */
    )
```

DESCRIPTION

This routine sends the message in *buffer* of length *nBytes* to the message queue *msgQId*. If any tasks are already waiting to receive messages on the queue, the message is immediately delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue and, if a task has previously registered to receive events from the message queue, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the message queue fails to send events, and if it was created using the **MSG_Q_EVENTSEND_ERR_NOTIFY** option, **ERROR** is returned even though the message was successfully sent to the queue.

The *timeout* parameter specifies the number of ticks to wait for adding its message to the queue if the message queue is full. The *timeout* parameter can also have the following special values:

NO_WAIT (0)
Return immediately, even if the message has not been sent.

WAIT_FOREVER (-1)
Never time out.

The *priority* parameter specifies the priority of the message being sent. The possible values are:

MSG_PRI_NORMAL (0)
Normal priority; add the message to the tail of the list of queued messages.

MSG_PRI_URGENT (1)
Urgent priority; add the message to the head of the list of queued messages.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an ISR and a task. When called from an ISR, *timeout* must be **NO_WAIT**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK on success or **ERROR** otherwise

ERRNO

S_smObjLib_NOT_INITIALIZED
The shared memory message queue library (VxMP Option) was not initialized.

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid.

S_objLib_OBJ_DELETED
The message queue was deleted while the calling task was pended.

S_objLib_OBJ_UNAVAILABLE
No free buffer space was available and the **NO_WAIT** timeout was specified.

S_objLib_OBJ_TIMEOUT
A timeout occurred while waiting for buffer space to become available.

S_msgQLib_INVALID_MSG_LENGTH
The message length exceeds the limit.

S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
The routine was called from an ISR with a non-zero timeout.

S_eventLib_EVENTSEND_FAILED
The message queue failed to send events to the registered task. This **errno** value can occur only if the message queue was created with the **MSG_Q_EVENTSEND_ERR_NOTIFY** option.

SEE ALSO

msgQLib, **msgQSmLib**, **msgQEvStart( )**

msgQShow( )

NAME

msgQShow( ) – show information about a message queue

SYNOPSIS

```
STATUS msgQShow
    (
    MSG_Q_ID msgQId,  /* message queue to display */
    int      level    /* 0 = summary, 1 = details */
    )
```

DESCRIPTION

This routine displays the state, and optionally the contents, of a message queue.

A summary of the state of the message queue is displayed as follows:

```
Message Queue Id    : 0x3f8c20
Task Queuing        : FIFO
Message Byte Len    : 150
Messages Max        : 50
Messages Queued     : 0
Receivers Blocked   : 1
Send timeouts       : 0
Receive timeouts    : 0
Options             : 0x1       MSG_Q_FIFO

VxWorks Events
--------------
Registered Task     : 0x3f5c70 (t1)
Event(s) to Send    : 0x1
Options             : 0x7       EVENTS_SEND_ONCE
                                EVENTS_ALLOW_OVERWRITE
                                EVENTS_SEND_IF_FREE
```

If *level* is 1, then more detailed information is displayed. If messages are queued, they are displayed as follows:

```
Messages queued:
  #    address length value
  1 0x123eb204    4   0x00000001 0x12345678
```

If tasks are blocked on the queue, they are displayed as follows:

```
Receivers blocked:

   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask     3fd678   0   21
```

RETURNS **OK** or **ERROR**.

ERRNO **S_smObjLib_NOT_INITIALIZED**
The shared memory message queue library (VxMP Option) was not initialized.

SEE ALSO **msgQShow**, **windsh**

msgQShowInit( )

NAME **msgQShowInit( )** – initialize the message queue show facility

SYNOPSIS

```
void msgQShowInit (void)
```

DESCRIPTION This routine 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:

- Using the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.
- Using the project facility, select **INCLUDE_MSG_Q_SHOW**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **msgQShow**

msgQSmCreate( )

NAME **msgQSmCreate( )** – create and initialize a shared memory message queue (VxMP Option)

SYNOPSIS

```
MSG_Q_ID msgQSmCreate
    (
    int maxMsgs,        /* max messages that can be queued */
    int maxMsgLength,   /* max bytes in a message */
    int options         /* message queue options */
    )
```

DESCRIPTION This routine creates a shared memory message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. It returns a message queue ID used to identify the created message queue. The queue can only be created with the option **MSG_Q_FIFO** (0), thus queuing pended tasks in FIFO order.

The global message queue identifier returned can be used directly by generic message queue handling routines in **msgQLib** -- **msgQSend( )**, **msgQReceive( )**, and **msgQNumMsgs( )** -- and by the show routines **show( )** and **msgQShow( )**.

If there is insufficient memory to store the message queue structure in the shared memory message queue partition or if the shared memory system pool cannot handle the requested message queue size, shared memory message queue creation will fail with **errno** set to **S_memLib_NOT_ENOUGH_MEMORY**. This problem can be solved by incrementing the value of **SM_OBJ_MAX_MSG_Q** and/or the shared memory objects dedicated memory size **SM_OBJ_MEM_SIZE** .

Before this routine can be called, the shared memory objects facility must be initialized (see **msgQSmLib**).

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **MSG_Q_ID**, or **NULL** if error.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine has been called from ISR.

S_objLib_OBJ_ID_ERROR
The shared memory message queue partition has not been initialized properly.

S_memLib_NOT_ENOUGH_MEMORY
Can't allocate shared memory message queue object.

S_msgQLib_INVALID_QUEUE_TYPE
Incorrect message queue pend queue type specified.

S_msgQLib_INVALID_MSG_COUNT
Incorrect number (negative) of messages specified.

S_msgQLib_INVALID_MSG_LENGTH
Incorrect length (negative) of messages specified.

S_smObjLib_LOCK_TIMEOUT
Can't get the lock on the shared memory message queue partition in time.

SEE ALSO **msgQSmLib**, **smObjLib**, **msgQLib**, **msgQShow**

msgQUnlink( )

NAME **msgQUnlink( )** – unlink a named message queue

SYNOPSIS

```
STATUS msgQUnlink
    (
    const char * name  /* name of message queue to unlink */
    )
```

DESCRIPTION This routine removes a message queue from the name space, and marks it as ready for deletion on the last **msgQClose( )**. In the case where there is no outstanding **msgQOpen( )** call, the message queue is deleted immediately.

After a message queue is unlinked, subsequent calls to **msgQOpen( )** using *name* will not be able to find the message queue, even if it has not been deleted yet. Instead, a new message queue could be created if **msgQOpen( )** is called with the **OM_CREATE** flag.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL** or empty.

S_objLib_OBJ_NOT_FOUND
No message queue with *name* was found.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The message queue is not named.

S_objLib_OBJ_DESTROY_ERROR
An error was detected while deleting the message queue.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **msgQOpen**, **msgQOpen( )**, **msgQClose( )**

munlock( )

NAME **munlock( )** – unlock specified pages (POSIX)

SYNOPSIS

```
int munlock
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION This routine unlocks specified pages from being memory resident.

RETURNS 0 (**OK**) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

munlockall( )

NAME **munlockall( )** – unlock all pages used by a process (POSIX)

SYNOPSIS

```
int munlockall (void)
```

DESCRIPTION This routine unlocks all pages used by a process from being memory resident.

RETURNS 0 (**OK**) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

mv( )

NAME **mv( )** – mv file into other directory.

SYNOPSIS

```
STATUS mv
    (
    const char * src,  /* source file name or wildcard */
    const char * dest  /* destination name or directory */
    )
```

DESCRIPTION This function is similar to **rename( )** but behaves somewhat more like the UNIX program "mv", it will overwrite files.

This command moves the *src* file or directory into a file which name is passed in the *dest* argument, if *dest* is a regular file or does not exist. If *dest* name is a directory, the source object is moved into this directory as with the same name, if *dest* is **NULL**, the current directory is assumed as the destination directory. *src* may be a single file name or a path containing a wildcard pattern, in which case all files or directories matching the pattern will be moved to *dest* which must be a directory in this case.

EXAMPLES

```
-> mv( "/sd0/dir1","/sd0/dir2")
-> mv( "/sd0/*.tmp","/sd0/junkdir")
-> mv( "/sd0/FILE1.DAT","/sd0/dir2/f001.dat")
```

RETURNS **OK** or error if any of the files or directories could not be moved, or if *src* is a pattern but the destination is not a directory.

ERRNO Not Available

SEE ALSO **usrFsLib**, the VxWorks programmer guides.

nanosleep( )

NAME

nanosleep( ) – suspend the current task until the time interval elapses (POSIX)

SYNOPSIS

```
int nanosleep
    (
    const struct timespec * rqtp,  /* time to delay                       */
    struct timespec       * rmtp   /* premature wakeup (NULL=no result) */
    )
```

DESCRIPTION

This routine suspends the current task for a specified time *rqtp* or until a signal or event notification is made.

The suspension may be longer than requested due to the rounding up of the request to the timer's resolution or to other scheduling activities (e.g., a higher priority task intervenes).

The **timespec** structure is defined as follows:

```
struct timespec
    {
                               /* interval = tv_sec*10**9 + tv_nsec */
     time_t tv_sec;            /* seconds */
     long tv_nsec;             /* nanoseconds (0 - 1,000,000,000) */
     };
```

If *rmtp* is non-**NULL**, the **timespec** structure is updated to contain the amount of time remaining. If *rmtp* is **NULL**, the remaining time is not returned. The *rqtp* parameter is greater than 0 or less than or equal to 1,000,000,000.

RETURNS

0 (**OK**), or -1 (**ERROR**) if the routine is interrupted by a signal or an asynchronous event notification, or *rqtp* is invalid.

ERRNO

EINVAL

EINTR

SEE ALSO

timerLib, **sleep( )**, **taskDelay( )**

netHelp( )

NAME

netHelp( ) – print a synopsis of network routines

SYNOPSIS

```
void netHelp (void)
```

DESCRIPTION

This command prints a brief synopsis of network facilities that are typically called from the shell.

```
hostAdd      "hostname","inetaddr" - add a host to remote host table;
                                     "inetaddr" must be in standard
                                     Internet address format e.g. "90.0.0.4"
hostShow                           - print current remote host table
netDevCreate "devname","hostname",protocol
                                   - create an I/O device to access
                                     files on the specified host
                                     (protocol 0=rsh, 1=ftp)
routeAdd     "destaddr","gateaddr" - add route to route table
routeDelete  "destaddr","gateaddr" - delete route from route table
routeShow                          - print current route table
iam          "usr"[,"passwd"]      - specify the user name by which
                                     you will be known to remote
                                     hosts (and optional password)
whoami                             - print the current remote ID
rlogin       "host"                - log in to a remote host;
                                     "host" can be inet address or
                                     host name in remote host table

ifShow       ["ifname"]            - show info about network interfaces
inetstatShow                       - show all Internet protocol sockets
tcpstatShow                        - show statistics for TCP
udpstatShow                        - show statistics for UDP
ipstatShow                         - show statistics for IP
icmpstatShow                       - show statistics for ICMP
arptabShow                         - show a list of known ARP entries
mbufShow                           - show mbuf statistics

EXAMPLE:  -> hostAdd "wrs", "90.0.0.2"
          -> netDevCreate "wrs:", "wrs", 0
          -> iam "fred"
          -> copy <wrs:/etc/passwd   /* copy file from host "wrs" */
          -> rlogin "wrs"            /* rlogin to host "wrs"      */
```

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, the VxWorks programmer guides.

nfsAuthUnixGet( )

NAME **nfsAuthUnixGet( )** – get the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixGet
    (
    char *machname,  /* where to store host machine        */
    int  *pUid,      /* where to store user ID             */
    int  *pGid,      /* where to store group ID            */
    int  *pNgids,    /* where to store number of group IDs */
    int  *gids       /* where to store array of group IDs  */
    )
```

DESCRIPTION This routine gets the previously set UNIX authentication values.

RETURNS N/A

ERRNO Not Available

SEE ALSO **nfsCommon**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsIdSet( )**

nfsAuthUnixPrompt( )

NAME **nfsAuthUnixPrompt( )** – modify the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixPrompt (void)
```

DESCRIPTION This routine allows UNIX authentication parameters to be changed from the shell. The user is prompted for each parameter, which can be changed by entering the new value next to the current one.

EXAMPLE

```
-> nfsAuthUnixPrompt
machine name:   yuba
user ID:        2001 128
group ID:       100
num of groups:  1 3
group #1:        100 100
group #2:        0 120
group #3:        0 200
value = 3 = 0x3
```

RETURNS Not Available

ERRNO Not Available

SEE ALSO **nfsCommon**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

nfsAuthUnixSet( )

NAME

nfsAuthUnixSet( ) – set the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixSet
    (
    char *machname,  /* host machine         */
    int  uid,        /* user ID              */
    int  gid,        /* group ID             */
    int  ngids,      /* number of group IDs */
    int  *aup_gids   /* array of group IDs  */
    )
```

DESCRIPTION

This routine sets UNIX authentication parameters. It is initially called by **usrNetInit( )**. **machname** should be set with the name of the mounted system (i.e. the target name itself) to distinguish hosts from hosts on a NFS network.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

nfsCommon, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

nfsAuthUnixShow( )

NAME

nfsAuthUnixShow( ) – display the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixShow (void)
```

DESCRIPTION

This routine displays the parameters set by **nfsAuthUnixSet( )** or **nfsAuthUnixPrompt( )**.

EXAMPLE

```
   -> nfsAuthUnixShow
   machine name = yuba
   user ID      = 2001
   group ID     = 100
   group [0]    = 100
   value = 1 = 0x1
```

RETURNS

N/A

ERRNO

Not Available

SEE ALSO **nfsCommon**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

nfsChkFilePerms( )

NAME **nfsChkFilePerms( )** – check the NFS file permissions with a given permission.

SYNOPSIS

```
STATUS nfsChkFilePerms
    (
    int nfsPerms,  /* permissions of the opened file */
    int ruid,      /* Remote uid */
    int rgid,      /* Remote gid */
    int perm       /* permission to be checked for 4:read 2:write 1:execute
*/
    )
```

DESCRIPTION This routine compares the NFS file permissions with a given permission.

This routine is basically designed for **nfsOpen( )** to verify the target file's permission prior to deleting it due to **O_TRUNC**.

The parameter *perm* will take 4(read), 2(write), 1(execute), or combinations of them.

OK means the file has valid permission whichever group is.

Called only by the I/O system.

RETURNS **OK, ERROR**

ERRNO Not Available

SEE ALSO **nfsCommon**

nfsDevInfoGet( )

NAME **nfsDevInfoGet( )** – read configuration information from the requested device

SYNOPSIS

```
STATUS nfsDevInfoGet
    (
    unsigned long  nfsDevHandle,  /* NFS device handle */
    NFS_DEV_INFO * pnfsInfo       /* ptr to struct to hold config info */
    )
```

DESCRIPTION This routine accesses the NFS device specified in the parameter *nfsDevHandle* and fills in the structure pointed to by *pnfsinfo*. The calling function should allocate memory for *pnfsinfo* and for the two character buffers "remFileSys" and "locFileSys" , that are part of *pnfsInfo*. These buffers should have a size of nfsMaxPath

RETURNS **OK**, if *pnfsInfo* information is valid, otherwise **ERROR**

ERRNO **S_objLib_OBJ_UNAVAILABLE**
This operation is not supported by the available NFS versions.

SEE ALSO **nfsCommon**, **nfsDevListGet( )**

nfsDevListGet( )

NAME **nfsDevListGet( )** – create list of all the NFS devices in the system

SYNOPSIS

```
int nfsDevListGet
    (
    unsigned long nfsDevList[],  /* NFS dev list of handles */
    int           listSize       /* number of elements available in the list
*/
    )
```

DESCRIPTION This routine fills the array *nfsDevlist* up to *listSize*, with handles to NFS devices currently in the system.

RETURNS The number of entries filled in the *nfsDevList* array.

ERRNO N/A

SEE ALSO **nfsCommon**, **nfsDevInfoGet( )**

nfsDevShow( )

NAME **nfsDevShow( )** – display the mounted NFS devices

SYNOPSIS

```
void nfsDevShow (void)
```

DESCRIPTION This routine displays the device names and their associated NFS file systems.

EXAMPLE

```
-> nfsDevShow
device name          file system
-----------          -----------
/yuba1/              yuba:/yuba1
/wrs1/               wrs:/wrs1
```

RETURNS N/A

ERRNO Not Available

SEE ALSO **nfsCommon**

nfsDrvNumGet( )

NAME **nfsDrvNumGet( )** – Get driver number of NFS device

SYNOPSIS

```
STATUS nfsDrvNumGet
    (
    int version  /* Version number of NFS */
    )
```

DESCRIPTION This routine returns the NFS driver number for the version requested. If the user specifies NFS version 2, this routine will return the value stored in variable nfs2DrvNum. If the user specifies NFS version 3, this routine will return the value stored in the variable nfs3DrvNum.

If the NFS driver of the user-specified version is yet to be initialized, or if initialization failed, nfsDrvNumGet will return **ERROR**.

RETURNS Returns the NFS driver number or **ERROR**.

ERRNO **S_objLib_OBJ_UNAVAILABLE**
This version does not support this operation.

SEE ALSO **nfsCommon**

nfsErrnoSet( )

NAME **nfsErrnoSet( )** – set NFS status

SYNOPSIS

```
void nfsErrnoSet
```

```
    (
    enum nfsstat status
    )
```

DESCRIPTION nfsErrnoSet calls errnoSet with the given "nfs stat" or'd with the NFS status prefix.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **nfsCommon**

nfsExport( )

NAME **nfsExport( )** – specify a file system to be NFS exported

SYNOPSIS

```
STATUS nfsExport
    (
    char * directory,  /* Directory to export - FS must support NFS */
    int    id,         /* ID number for file system */
    BOOL   readOnly,   /* TRUE if file system is exported read-only */
    int    options     /* Reserved for future use - set to 0 */
    )
```

DESCRIPTION This routine makes a file system available for mounting by a client. The client should be in the local host table (see **hostAdd( )**), although this is not required.

The *id* parameter can either be set to a specific value, or to 0. If it is set to 0, an ID number is assigned sequentially. Every time a file system is exported, it must have the same ID number, or clients currently mounting the file system will not be able to access files.

NOTE exporting a file system requires at least 512kb of free space available on the file system for creation of configuration files.

To display a list of exported file systems, use:

```
    -> nfsExportShow "localhost"
```

RETURNS **OK**, or **ERROR** if the file system could not be exported.

ERRNO Not Available

SEE ALSO **mountd**, **nfsLib**, **nfsExportShow( )**, **nfsUnexport( )**

nfsExportShow( )

NAME **nfsExportShow( )** – display the exported file systems of a remote host

SYNOPSIS

```
STATUS nfsExportShow
    (
    char *hostName  /* host machine to show exports for */
    )
```

DESCRIPTION This routine displays the file systems of a specified host and the groups that are allowed to mount them.

EXAMPLE

```
   -> nfsExportShow "wrs"
   /d0               staff
   /d1               staff eng
   /d2               eng
   /d3
   value = 0 = 0x0
```

RETURNS **OK** or **ERROR**.

ERRNO

S_hostLib_INVALID_PARAMETER
hostName is invalid.

S_objLib_OBJ_UNAVAILABLE
This operation is not supported by the available NFS versions.

S_nfsLib_NFSERR_NOTSUPP
Remote system does not have a compatible mount version.

SEE ALSO **nfsCommon**

nfsHelp( )

NAME **nfsHelp( )** – display the NFS help menu

SYNOPSIS

```
void nfsHelp (void)
```

DESCRIPTION This routine displays a summary of NFS facilities typically called from the shell:

```
    nfsHelp                        Print this list
    netHelp                        Print general network help list
    nfsMount "host","filesystem"[,"devname"]  Create device with
                                     file system/directory from host
    nfsUnmount "devname"           Remove an NFS device
```

```
nfsAuthUnixShow               Print current UNIX authentication
nfsAuthUnixPrompt             Prompt for UNIX authentication
nfsIdSet id                   Set user ID for UNIX authentication
nfsDevShow                    Print list of NFS devices
nfsExportShow "host"          Print a list of NFS file systems which
                                are exported on the specified host
mkdir "dirname"               Create directory
rm "file"                     Remove file

EXAMPLE:  -> hostAdd "wrs", "90.0.0.2"
          -> nfsMount "wrs","/disk0/path/mydir","/mydir/"
          -> cd "/mydir/"
          -> nfsAuthUnixPrompt     /* fill in user ID, etc.     */
          -> ls                    /* list /disk0/path/mydir    */
          -> copy < foo            /* copy foo to standard out  */
          -> ld < foo.o            /* load object module foo.o  */
          -> nfsUnmount "/mydir/"  /* remove NFS device /mydir/ */
```

RETURNS N/A

ERRNO Not Available

SEE ALSO **nfsCommon**

nfsIdSet( )

NAME **nfsIdSet( )** – set the ID number of the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsIdSet
    (
    int uid  /* user ID on host machine */
    )
```

DESCRIPTION This routine sets only the UNIX authentication user ID number. For most NFS permission needs, only the user ID needs to be changed. Set *uid* to the user ID on the NFS server.

RETURNS N/A

ERRNO Not Available

SEE ALSO **nfsCommon**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**

nfsMntDump( )

NAME **nfsMntDump( )** – display all NFS file systems mounted on a particular host

SYNOPSIS

```
STATUS nfsMntDump
    (
    const char *hostName  /* host machine */
    )
```

DESCRIPTION This routine displays all the NFS file systems mounted on a specified host machine.

RETURNS **OK** or **ERROR**.

ERRNO

S_nfsLib_NFSERR_INVAL
hostName is invalid.

S_nfsLib_NFSERR_NOTSUPP
Remote system does not have a compatible mount version.

S_objLib_OBJ_UNAVAILABLE
This routine is not supported by the included NFS versions.

SEE ALSO **nfsCommon**

nfsMount( )

NAME **nfsMount( )** – mount an NFS file system

SYNOPSIS

```
STATUS nfsMount
    (
    const char *host,        /* name of remote host */
    const char *fileSystem,  /* name of remote directory to mount */
    const char *localName    /* local device name for remote dir */
    )
```

DESCRIPTION This routine mounts a remote file system. It creates a local device *localName* for a remote file system on a specified host. The host must have already been added to the local host table with **hostAdd( )**.

RETURNS **OK**, or **ERROR** if the driver is not installed, *host* is invalid, or memory is insufficient.

ERRNO

S_nfsLib_NFSERR_INVAL
Provided arguments are invalid.

S_objLib_OBJ_UNAVAILABLE
This operation is not supported by the included NFS versions.

S_nfsLib_NFSERR_NOTSUPP
Remote system does not have a compatible mount version.

SEE ALSO **nfsCommon**, **nfsUnmount( )**, **hostAdd( )**

nfsMountAll( )

NAME **nfsMountAll( )** – mount all file systems exported by a specified host

SYNOPSIS

```
STATUS nfsMountAll
    (
    const char *pHostName,    /* name of remote host */
    const char *pClientName,  /* name of a client specified in access list,
if any */
    BOOL       quietFlag      /* FALSE = print name of each mounted file
system */
    )
```

DESCRIPTION This routine mounts the file systems exported by the host *pHostName* which are accessible by *pClientName*. A *pClientName* entry of **NULL** will only mount file systems that are accessible by any client. The **nfsMount( )** routine is called to mount each file system. It creates a local device for each mount that has the same name as the remote file system.

If the *quietFlag* setting is **FALSE**, each file system is printed on standard output after it is mounted successfully.

RETURNS **OK**, or **ERROR** if any mount fails.

ERRNO **S_nfsLib_NFSERR_INVAL**
Invalid arguments.

S_objLib_OBJ_UNAVAILABLE
This operation is not supported by the included NFS versions.

S_nfsLib_NFSERR_NOTSUPP
Incompatible mount version on remote system.

SEE ALSO **nfsCommon**, **nfsMount( )**

nfsStatusGet( )

NAME

nfsStatusGet( ) – Get the statistics of the NFS server

SYNOPSIS

```
STATUS nfsStatusGet
    (
    void * serverStats,  /* pointer to status struct */
    int    version       /* NFS v2 or NFS v3 */
    )
```

DESCRIPTION

This routine returns the statistics of the NFS procedure calls made by the remote NFS clients *serverStats* pointer to a memory location where the statistics information will be copied. *version* Statistics of which NFS version is desired. This parameter takes two values only. 0x01 for NFS Version 2 and 0x02 for NFS version 3.

RETURNS

OK or **ERROR** if version is invalid.

ERRNO

Not Available

SEE ALSO

nfsdCommon

nfsUnexport( )

NAME

nfsUnexport( ) – remove a file system from the list of exported file systems

SYNOPSIS

```
STATUS nfsUnexport
    (
    char * dirName  /* Name of the directory to unexport */
    )
```

DESCRIPTION

This routine removes a file system from the list of file systems exported from the target. Any client attempting to mount a file system that is not exported will receive an error (**NFSERR_ACCESS**).

RETURNS

OK, or **ERROR** if the file system could not be removed from the exports list.

ERRNO

ENOENT

SEE ALSO

mountd, **nfsLib**, **nfsExportShow( )**, **nfsExport( )**

nfsUnmount( )

NAME

nfsUnmount( ) – unmount an NFS device

SYNOPSIS

```
STATUS nfsUnmount
    (
    const char *localName  /* local of nfs device */
    )
```

DESCRIPTION

This routine unmounts file systems that were previously mounted via NFS.

RETURNS

OK, or **ERROR** if *localName* is not an NFS device or cannot be mounted.

ERRNO

S_nfsLib_NFSERR_INVAL
localName is invalid.

S_nfsDrv_NOT_AN_NFS_DEVICE
localName is not an NFS device.

S_objLib_OBJ_UNAVAILABLE
This operation is not supported by the included NFS versions.

SEE ALSO

nfsCommon, **nfsMount( )**

nfsdHashTableParamsSet( )

NAME

nfsdHashTableParamsSet( ) – sets up the parameters for the NFS hash table

SYNOPSIS

```
void nfsdHashTableParamsSet
    (
    int    bucketSize,
    int    tableLen,
    char * basePath
    )
```

DESCRIPTION

This function must be called prior to the call to nfsExport, and can be called repeatedly to configure different exports differently.

bucketSize represents the number of bytes available in each hash bucket. The selectable sizes are 512, 1k, 2k, 4k, 8k, 16k. It is probably a good idea to make this value the same size as a disk allocation unit in general. Never smaller, but perhaps larger if you plan on accessing lots of files with very long filenames. Note, that the code always keeps one bucket in memory, and reads and writes whole buckets at a time. So larger buckets can impact performance. A value of zero preserves the previous value. The default is 1k.

tableLen represents the number of buckets. As we all know, prime numbers work best for hashing, so selectable sizes for *tableLen* are all prime numbers that are close to powers of 2. Specifically: 7, 17, 31, 61, 127, 251, 509, 1021, and 2039. When the hash-function is applied to a filename or inode, it is the result of the modulus operation that then determines the bucket the value is placed into. A value of zero preserves the previous value. The default value is 509.

basePath allows you to move the hash file to a seperate location. Perhaps a disk volume dedicated to hash files, or a RAM drive if you do not need the inode-to-filename mappings preserved. The default is "" which indicates that the hash files will be saved to the exported volume. nfsExport("/foo",0,0) would result in /foo/nfsHashTbl.cfg and /foo/nfsHashTbl.cfg being created. With basePath set to "/**ramDrv**", the files would be placed in the /**ramDrv** volume and pre-pended with the export path. Specifically, the files created would be /ramDrv/foo_nfsHashTbl.cfg and /ramDrv/foo_nfsHashTbl.dat. Passing in **NULL** preserves the existing value. The default value is "";

bucketSize and *tableLen* are only used when creating a hash table for the first time. After that, their values are preserved in nfsHashTbl.cfg. However, basePath (if different from the default) must be called so the NFS server can locate the files.

RETURNS
Not Available

ERRNO
Not Available

SEE ALSO
nfsHash

nfsdInit( )

NAME
nfsdInit( ) – initialize the NFS server

SYNOPSIS

```
STATUS nfsdInit
    (
    int     nServers,       /* the number of NFS servers to create */
    int     nExportedFs,    /* maximum number of exported file systems */
    int     priority,       /* the priority for the NFS servers */
    FUNCPTR authHook,       /* Authentication hook */
    FUNCPTR mountAuthHook,  /* authentication hook for mount daemon */
    int     options         /* 3 bits used only */
    )
```

DESCRIPTION
This routine initializes the NFS server. *nServers* specifies the number of Tasks to be spawned to handle NFS requests. *priority* is the priority that those tasks will run at. *authHook* is a pointer to an authorization routine. *mountAuthHook* is a pointer to a similar routine, passed to **mountdInit( )**. *options* (only 3 LSBs are used for specifying the NFS version). Currently *options* can take the following values. 0x01 to start the NFS V2 service

only. 0x02 to start the NFS V3 service only. 0x00 to start the NFS V2 and V3 services (default)

Normally, no authorization is performed by either mountd or nfsd. If you want to add authorization, set *authHook* to a function pointer to a routine declared as follows:

```
nfsstat routine
    (
    int                progNum,       /* RPC program number */
    int                versNum,       /* RPC program version number */
    int                procNum,       /* RPC procedure number */
    struct sockaddr_in clientAddr,    /* address of the client */
    NFSD_ARGUMENT *    nfsdArg        /* argument of the call */
    )
```

The nfsdArg will be of type "**NFSD_ARGUMENT**" if versNum is 2. The nfsdArg will be of type "NFS3D_ARGUMENT" if versNum is 3. The user authentication hook must use the nfsdArg accordingly.

The *authHook* routine should return **NFS_OK** if the request is authorized, and NFS3ERR_ACCES if not. (**NFSERR_ACCESS** is not required; any legitimate NFS error code can be returned.)

See **mountdInit( )** for documentation on *mountAuthHook*. Note that *mountAuthHook* and *authHook* can point to the same routine. Simply use the *progNum*, *versNum*, and *procNum* fields to decide whether the request is an NFS request or a mountd request.

RETURNS

OK, or **ERROR** if the NFS server cannot be started.

ERRNO

Not Available

SEE ALSO

nfsd, **nfsExport( )**, **mountdInit( )**

nfsdStatusShow( )

NAME

nfsdStatusShow( ) – show the status of the NFS server

SYNOPSIS

```
STATUS nfsdStatusShow
    (
    int options  /* unused */
    )
```

DESCRIPTION

This routine shows statistics of procedure calls to the NFS server. This routine takes one parameter to specify the NFS server version whose statistics are to be shown. *options* takes one of the following 3 values. 0x01 Display statistics of NFS version 2 server only. 0x02 Display statistics of NFS version 3 server only. 0x00 Display statistics of NFS version 2 & 3.

RETURNS **OK**, or **ERROR** if the information cannot be obtained.

ERRNO Not Available

SEE ALSO **nfsdCommon**

nicRegister( )

NAME **nicRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void nicRegister(void)
```

DESCRIPTION This routine registers the ST-NIC driver with VxBus as a child of the PLB bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **ns83902VxbEnd**

npc( )

NAME **npc( )** – return the contents of the next program counter (SimSolaris)

SYNOPSIS

```
int npc
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the next program counter from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the next program counter.

ERRNO Not Available

SEE ALSO **dbgArchLib**, **ti( )**

nseRegister( )

NAME
nseRegister( ) – register with the VxBus subsystem

SYNOPSIS
```
void nseRegister(void)
```

DESCRIPTION
This routine registers the NatSemi driver with VxBus as a child of the PCI bus type.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
ns8381xVxbEnd

nvRamSegDefGet( )

NAME
nvRamSegDefGet( ) – get segment allocation from BSP

SYNOPSIS
```
STATUS nvRamSegDefGet
    (
    VXB_DEVICE_ID    pInst,
    HCF_DEVICE *     pHcf,
    NVRAM_SEGMENT ** ppSegList,
    int *            pSegListCount
    )
```

DESCRIPTION
This routine reads entries from the hcf record for the NVRam device specified by pInst, and fills them in to a table.

Allocation of the table is performed within this routine, using **hwMemLib** allocation. The table consists of a nSeg field indicating the number of segments, followed by a set of structures containing the start offset, size, allocation name, and allocation unit number.

Within the **hwconf.c** file, resource names are specified with the name "segAddr". Segment size names are "segSz". Driver (or OS module) names are "drvName". Driver unit number names are "drvUnit". Note that each of these four resource names must be specified for each segment, regardless of whether the segment is allocated or not. For segments not allocated, the segment should

either be omitted, or allocated to a module named "unallocated"
unit -1.

The hwconf entries for each segment must be grouped together,
and each segment must have all segments fully described: name,
unit, addr, size. Failure to meet these restrictions will result
in segment information being corrupted or discarded.

RETURNS **OK**, or **ERROR** if the table cannot be allocated

ERRNO Not Available

SEE ALSO **vxbNonVolLib**

o0( )

NAME **o0( )** – return the contents of register o0 (also o1-o7) (SimSolaris)

SYNOPSIS

```
int o0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of out register o0 from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all out registers (o0 - o7): **o0( )** - **o7( )**.

The stack pointer is accessed via o6.

RETURNS The contents of register o0 (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

objClassTypeGet( )

NAME

objClassTypeGet( ) – get an object's class type

SYNOPSIS

```
enum windObjClassType objClassTypeGet
    (
    OBJ_ID objId
    )
```

DESCRIPTION

The class type of the specified object is returned.

RETURNS

class type enum, or **ERROR** if caller has insufficient access rights.

ERRNO

Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

SEE ALSO

objLib

objContextGet( )

NAME

objContextGet( ) – return the object's context value

SYNOPSIS

```
STATUS objContextGet
    (
    OBJ_ID  objId,    /* object to get context from */
    void ** pContext  /* where to store context value */
    )
```

DESCRIPTION

The value stored in the object's context field is returned. The context field is typically set by calling **objContextSet( )** or an **xxxOpen( )** routine.

RETURNS

OK, or **ERROR** if objId is invalid.

ERRNO

Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

SEE ALSO

objLib

objContextSet( )

NAME

objContextSet( ) – set the object's context value

SYNOPSIS

```
STATUS objContextSet
    (
    OBJ_ID objId,    /* object to set context on */
    void * context   /* context value */
    )
```

DESCRIPTION

This routine sets the object's context field. This value is not actually used by VxWorks. Instead, the context value can be used by OS extensions to implement object permissions, for example.

RETURNS

OK, or **ERROR** if objId is invalid.

ERRNO

Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

SEE ALSO

objLib

objHandleShow( )

NAME

objHandleShow( ) – show information on the object referenced by an object handle

SYNOPSIS

```
STATUS objHandleShow
    (
    OBJ_HANDLE objHandle,  /* object handle to get information from */
    RTP_ID     rtpId       /* ID of RTP to which objHandle belongs */
    )
```

DESCRIPTION

This routine displays information regarding the WIND object an object handle references. This routine is intended to be used for debugging purposes.

RETURNS

OK, or **ERROR** if the information could not be displayed.

ERRNO

N/A

SEE ALSO

objShow, **objShowAll( )**

objHandleTblShow( )

NAME **objHandleTblShow( )** – show information on an RTP's handle table

SYNOPSIS

```
void objHandleTblShow
    (
    RTP_ID rtpId,
    int    disp
    )
```

DESCRIPTION This routine displays the contents of the supplied *rtpId* handle table. The argument *count* indicates the number of slots in the table to display. In case *count* is zero, all the in-use slots in the table are displayed. This routine is intended to be used only for debugging purposes.

RETURNS N/A

ERRNO N/A

SEE ALSO **objShow**, **rtpShow( )**, **rtpDetailShow( )**

objNameGet( )

NAME **objNameGet( )** – get an object's name

SYNOPSIS

```
STATUS objNameGet
    (
    OBJ_ID objId,     /* pointer to object to get name */
    char * nameBuf,   /* pointer to name string buffer */
    int    bufSize    /* size, in bytes, of name buffer */
    )
```

DESCRIPTION The specified object's name string is copied into *nameBuf*.

RETURNS **OK**, or **ERROR** if object name cannot be retrieved.

ERRNO Possible errnos generated by this routine include:

S_objLib_OBJ_NAME_TRUNCATED
Supplied name buffer is too small. Truncated name has been returned.

S_objLib_OBJ_NOT_NAMED
Object has not been labeled with a name.

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

SEE ALSO **objLib**

objNameLenGet( )

NAME **objNameLenGet( )** – get an object's name length

SYNOPSIS
```
int objNameLenGet
    (
    OBJ_ID objId
    )
```

DESCRIPTION The specified object's name length (without the terminating \0 character) is returned.

RETURNS name length or -1 if the object name cannot be retrieved

ERRNO Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Object class does not support the name get operation.

S_objLib_OBJ_NOT_NAMED
Object has not been labeled with a name.

SEE ALSO **objLib**

objNameToId( )

NAME **objNameToId( )** – find object with matching name string and type

SYNOPSIS
```
OBJ_ID objNameToId
    (
    enum windObjClassType classType,
    const char *          name
    )
```

DESCRIPTION

The object name space is searched for an object with a matching *name* and *classType*. There may exist more than one object of the same type with identical names. In such cases, this routine will return the id of the first object found.

This routine is provided if the **INCLUDE_OBJ_INFO** component is present in the configuration.

Values for the windObjClassType enumerated type are tabulated below:

Value	Object Class
0	Invalid
1	Wind Semaphore
2	POSIX Semaphore
3	Wind Message Queue
4	POSIX Message Queue
5	Real Time Process
6	Task
7	Watchdog Timer
8	File Descriptor
9	Page Pool
10	Page Manager
11	Group
12	Virtual Memory Context
13	Event Trigger
14	Memory Partition
15	I2O
16	device management system
17	Set
18	ISR object
19	POSIX Timer
20	Shared data region

RETURNS

NULL if no match occurs, otherwise **OBJ_ID** of matching object.

ERRNO

Possible errnos generated by this routine include:

S_objLib_OBJ_ILLEGAL_CLASS_TYPE
The specified object class type is invalid.

SEE ALSO

objLib

objOwnerGet( )

NAME
objOwnerGet( ) – return the object's owner

SYNOPSIS
```
OBJ_ID objOwnerGet
    (
    OBJ_ID objId  /* object to get owner from */
    )
```

DESCRIPTION
The ID of the object that owns the specified object is returned.

RETURNS
owner object ID, or **NULL** if invalid object id, the task does not
    have access rights, or object ownership is excluded from the system

ERRNO
Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

SEE ALSO
objLib

objOwnerSet( )

NAME
objOwnerSet( ) – change the object's owner

SYNOPSIS
```
STATUS objOwnerSet
    (
    OBJ_ID objId,   /* object to set owner */
    OBJ_ID ownerId  /* owner object ID */
    )
```

DESCRIPTION
Set the owner of an object. This routine is used to change the default object ownership hierarchy. The calling task must have access rights to both the object *objId* whose owner is being changed, and the owner object *ownerId* which must be a real time process.

The owner object *ownerId* must be a real time process.

If **INCLUDE_OBJ_OWNERSHIP** is excluded this routine simply returns **OK**.

RETURNS
ERROR if object ID or owner ID is invalid.

ERRNO
Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object or owner object identifier.

S_objLib_OBJ_INVALID_OWNER
Invalid object ownership relationship.

SEE ALSO **objLib**

objShow( )

NAME **objShow( )** – show information on an object

SYNOPSIS
```
STATUS objShow
    (
    OBJ_ID objId,    /* object to show information on */
    int    showType  /* show type */
    )
```

DESCRIPTION Call class attached show routine for an object.

RETURNS **OK**, or **ERROR** if information could not be shown.

ERRNO Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
Invalid object identifier.

S_objLib_OBJ_NO_METHOD
Show routine for this class of object not installed.

SEE ALSO **objLib**

objShowAll( )

NAME **objShowAll( )** – show all information on an object

SYNOPSIS
```
STATUS objShowAll
    (
    OBJ_ID objId,    /* object to show information on */
    int    showType  /* show type */
    )
```

DESCRIPTION

This routine displays all information about an object. The generic object information is handled directly by this routine, while the class specific information is handled by the show routine registered for the class. The routine **objShow( )** only displays the class-specific information. The *showType* parameter is passed transparently to the class-specific show routine. Typically, a *showType* of 1 is used to enable a detailed information display. If *objId* is not given or is null, **objShowAll( )** displays all information on the system.

EXAMPLE

The following example shows information on a task with TID = 0x188d78 :

```
[vxKernel] -> objShowAll 0x188d78

Generic Object Information
==========================

Type  : Task
Name  : /pubTestTask
Attr  : 0xc1 (WIND_OBJ_NAME_DYNAMIC WIND_OBJ_PUBLIC WIND_OBJ_NAMED)
refCnt: 2
Ctx   : 0x0 (type = 1)

Owner Information
-----------------
ID    : 0x000badc4
Type  : Real Time Process
Name  : (null)


Object Handles opened on this object:

Object Handle              RTP
-------------   --------------------------
 0x1e001d       0xdc722c     helloworld.vxe


Owned Objects
-------------

Object Id  Object Type                               Object Name
---------- ---------------------------------------  ------------------------
0x001ff2d8 Binary Semaphore                          (null)
0x001ff288 Binary Semaphore                          (userTblSem)
0x001ff238 Mutex Semaphore                           wdMutexSem
0x001ff1b8 |-Watchdog                                (null)
0x001ff168 Message Queue                             (null)


Task Specific Information
=========================

NAME         ENTRY     TID      PRI STATUS   PC       SP      ERRNO DELAY
------------ --------- -------- --- -------- -------- ------- ----- -----
/pubTestTask 0x45345   c7cce4   100 PEND     419563   dc9fc8      0     0
 value = 0 = 0x0
```

The ownership hierarchy is shown by an indentation of the object type. In the above example, the watchdog object is directly owned by a semaphore object (wdMutexSem), which in turn is owned by the task object (testTask). Hence the watchdog object is indirectly owned by the task object (testTask).

For an object that is not named, if a symbol table entry whose value matches the object id exists, the symbol name, enclosed in brackets, will be displayed under the object name column. When an object is not named and an exact symbol table match does not exist, "(null)" will be displayed under the object name column.

WARNING Deleting *objId* while **objShowAll( )** is gathering information, can lead to unexpected results.

RETURNS **OK**, or **ERROR** if the information could not be displayed.

ERRNO N/A

SEE ALSO **objShow**

open( )

NAME **open( )** – open a file

SYNOPSIS

```
int open
    (
    const char *name,  /* name of the file to open                    */
    int         flags,  /* access control flag                         */
    int         mode    /* mode of file to create (UNIX chmod style) */
    )
```

DESCRIPTION This routine opens a file for reading, writing, or updating, and returns a file descriptor for that file. The arguments to **open( )** are the filename *name* and the type of access set in *flags* and a UNIX chmod-style file mode *mode*.

The parameter *flags* is set to one or a combination of the following access settings by bitwise OR operation for the duration of time the file is open. The following list is just a generic description of supported settings. Their availability and effect with or without combination among them change from device to device. Check the specific device manual for further details.

O_RDONLY
Open for reading only.

O_WRONLY
Open for writing only.

O_RDWR
Open for reading and writing.

O_CREAT
Create a file if not existing.

O_EXCL
Error on open if file exists and **O_CREAT** is also set.

O_SYNC
Write on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_DSYNC
Write on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_RSYNC
Read on the file descriptor complete at the same sync level as **O_DSYNC** and **O_SYNC** flags.

O_APPEND
If set, the file offset is set to the end of the file prior to each write. So writes are guaranteed at the end. It has no effect on devices other than the regular file system.

O_NONBLOCK
Non-blocking I/O if being set.

O_NOCTTY
Do not assign a ctty on this open, which does not cause the terminal device to become the controlling terminal for the process. Effective only on a terminal device.

O_TRUNC
Open with truncation. If the file exists and is a regular file, and the file is successfully opened, its length is truncated to 0. It has no effect on devices other than the regular file system.

In general, **open( )** can only open pre-existing devices and files. However, files can also be created with **open( )** by setting **O_CREAT** and perhaps some other like **O_RDWR** which depends on the file system implementation. In this case, the file is created with a UNIX chmod-style file mode, as indicated with the parameter *mode*. For example:

```
fd = open ("/usr/myFile", O_CREAT | O_RDWR, 0644);
```

Files, on dosFs volumes, can be opened with the **O_SYNC** flag indicating that each write should be immediately written to the backing media. This synchronizes the FAT and the directory entries.

NOTE
For more information about situations when there are no file descriptors available, see the reference entry for **iosInit( )**.

Also note that not all device drivers honor the flags or mode values when opening a file. Most simple devices simply ignore them and return an open file descriptor for both reading and writing. Read the device driver manual for information on this.

RETURNS A file descriptor number, or **ERROR** if a file name is not specified, the device does not exist, no file descriptors are available, or the driver returns **ERROR**.

ERRNO

ELOOP
Circular symbolic link, too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in path.

others
Other errors reported by device drivers.

SEE ALSO **ioLib**, **creat( )**

opendir( )

NAME **opendir( )** – open a directory for searching (POSIX)

SYNOPSIS

```
DIR *opendir
    (
    const char* dirName  /* name of directory to open */
    )
```

DESCRIPTION This routine opens the directory named by *dirName* and allocates a directory descriptor (DIR) for it. A pointer to the DIR structure is returned. The return of a **NULL** pointer indicates an error.

After the directory is opened, **readdir( )** is used to extract individual directory entries. Finally, **closedir( )** is used to close the directory.

WARNING For remote file systems mounted over **netDrv**, **opendir( )** fails, because the **netDrv** implementation strategy does not provide a way to distinguish directories from plain files. To permit use of **opendir( )** on remote files, use NFS rather than **netDrv**.

RETURNS A pointer to a directory descriptor, or **NULL** if there is an error.

ERRNO N/A.

SEE ALSO **dirLib**, **closedir( )**, **readdir( )**, **rewinddir( )**, **ls( )**

operator_delete( )

NAME **operator_delete( )** – default run-time support for memory deallocation (C++)

SYNOPSIS

```
extern void operator delete
    (
    void *pMem  /* pointer to dynamically-allocated object */
    )
```

DESCRIPTION This function provides the default implementation of operator delete. It returns the memory, previously allocated by operator new, to the VxWorks system memory partition.

RETURNS N/A

ERRNO Not Available

SEE ALSO **cplusLib**

operator_new( )

NAME **operator_new( )** – default run-time support for operator new (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n  /* size of object to allocate */
    ) throw (std::bad_alloc)
```

DESCRIPTION This function provides the default implementation of operator new. It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If there is no new-handler an exception of type "bad_alloc" is thrown.

THROWS std::bad_alloc if allocation failed.

RETURNS Pointer to new object.

ERRNO Not Available

SEE ALSO **cplusLib**

operator_new( )

NAME **operator_new( )** – default run-time support for operator new (nothrow) (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t          n,  /* size of object to allocate */
    const nothrow_t &   /* supply argument of "nothrow" here */
    ) throw ()
```

DESCRIPTION This function provides the default implementation of operator new (nothrow). It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails, a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If the new_handler throws a bad_alloc exception, the exception is caught and 0 is returned. If allocation fails and there is no new_handler 0 is returned.

RETURNS Pointer to new object or 0 if allocation fails.

ERRNO Not Available

SEE ALSO **cplusLib**

operator_new( )

NAME **operator_new( )** – run-time support for operator new with placement (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n,    /* size of object to allocate (unused) */
    void * pMem  /* pointer to allocated memory         */
    )
```

DESCRIPTION This function provides the default implementation of the global new operator, with support for the placement syntax. New-with-placement is used to initialize objects for which memory has already been allocated. *pMem* points to the previously allocated memory. memory.

RETURNS *pMem*

ERRNO Not Available

SEE ALSO **cplusLib**

oprintf( )

NAME **oprintf( )** – write a formatted string to an output function

SYNOPSIS

```
int oprintf
    (
    FUNCPTR      prtFunc,  /* pointer to output function */
    int          prtArg,   /* argument for output function */
    const char * fmt,      /* format string to write */
    ...                    /* optional arguments to format string */
    )
```

DESCRIPTION This routine prints a formatted string via the function specified by *prtFunc*. The function will receive as parameters a pointer to a buffer, an integer indicating the length of the buffer, and the argument *prtArg*. If **NULL** is specified as the output function, the output will be sent to stdout.

The function and syntax of oprintf are otherwise identical to **printf( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The number of characters output, not including the **NULL** terminator.

ERRNO Not Available

SEE ALSO **fioBaseLib**, **printf( )**

partLibCreate( )

NAME

partLibCreate( ) – partition a device

SYNOPSIS

```
STATUS partLibCreate
    (
    int fd,
    int nPart,
    int size1,
    int size2,
    int size3
    )
```

DESCRIPTION

This routine partitions a device.

RETURNS

OK on success, **ERROR** otherwise

ERRNO

Not Available

SEE ALSO

partLib

passFsDevInit( )

NAME

passFsDevInit( ) – associate a device with passFs file system functions

SYNOPSIS

```
void * passFsDevInit
    (
    char * devName  /* device name */
    )
```

DESCRIPTION

This routine associates the name *devName* with the file system and installs it in the I/O System's device table.

RETURNS

A pointer to the volume descriptor on success, else **NULL**.

ERRNO

S_iosLib_DUPLICATE_DEVICE_NAME (**EINVAL**)
Device name already in use.

SEE ALSO

passFsLib

passFsInit( )

NAME

passFsInit( ) – prepare to use the passFs library

SYNOPSIS

```
STATUS passFsInit
    (
    int  passfs,      /* number of pass-through file systems  */
    BOOL cacheEnable  /* enable passfs cache ?                */
    )
```

DESCRIPTION

This routine initializes the passFs library. It must be called only once, before any other routines in the 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. This routine installs **passFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores.

Usually this routine is called from the root task, **usrRoot( )**, in **prjConfig( )**. This initialization is enabled when the configuration component **INCLUDE_PASSFS** is defined.

NOTE

Maximum number of pass-through file systems is 1.

RETURNS

OK on success, else **ERROR**.

ERRNO

S_iosLib_DRIVER_GLUT (**ENOMEM**)
No memory available for data structures.

SEE ALSO

passFsLib

pathconf( )

NAME

pathconf( ) – determine the current value of a configurable limit

SYNOPSIS

```
long pathconf
    (
    const char *path,  /* path of the file */
    int        name    /* Value to query */
    )
```

DESCRIPTION

The **fpathconf( )** and **pathconf( )** functions provide a method for the application to determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

RETURNS The current value is returned if valid with the query. Otherwise, **ERROR**, -1 returned and errno may be set to indicate the error. There are many reasons to return **ERROR**. If the variable corresponding to name has no limit for the path or file descriptor, both **pathconf( )** and **fpathconf( )** return -1 without changing errno.

ERRNO

SEE ALSO **fsPxLib**, **fpathconf( )**

pause( )

NAME **pause( )** – suspend the task until delivery of a signal (POSIX)

SYNOPSIS

```
int pause (void)
```

DESCRIPTION This routine suspends the task until delivery of a signal.

NOTE Since the **pause( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**

SEE ALSO **sigLib**

pc( )

NAME **pc( )** – return the contents of the program counter

SYNOPSIS

```
int pc
    (
    int task  /* task ID */
    )
```

DESCRIPTION This command extracts the contents of the program counter for a specified task from the task's TCB. If *task* is omitted or 0, the current task is used.

RETURNS the contents of the program counter.

ERRNO N/A

SEE ALSO **usrLib**, **ti( )**, the VxWorks programmer guides.

pcConDevBind( )

NAME **pcConDevBind( )** – bind keyboard or VGA device with console

SYNOPSIS

```
TY_DEV * pcConDevBind
    (
    int     arg,
    FUNCPTR pFunc,
    void *  pArg
    )
```

DESCRIPTION This routine is called by the keyboard and VGA drivers to associate themselves with a PC console instance. The keyboard driver should normally pass its unit number for arg, and **NULL** for the remaining two parameters. The VGA driver should pass a pointer to its buffer processing routine and the argument to this routine for these two parameters.

RETURNS **TY_DEV** pointer associated with console, or **NULL**.

ERRNO Not Available

SEE ALSO **vxbPcConsole**

pcConDevCreate( )

NAME **pcConDevCreate( )** – create a device for the on-board ports

SYNOPSIS

```
STATUS pcConDevCreate
    (
    char *   name,       /* name to use for this device  */
    FAST int channel,    /* virtual console number       */
    int      rdBufSize,  /* read buffer size, in bytes   */
    int      wrtBufSize  /* write buffer size in bytes   */
    )
```

DESCRIPTION This routine creates a device on one of the pcConsole ports. Each port to be used should have only one device associated with it, by calling this routine.

RETURNS **OK**, or **ERROR** if there is no driver or one already exists for the specified port.

ERRNO Not Available

SEE ALSO **vxbPcConsole**

pcConDrv( )

NAME **pcConDrv( )** – initialize the console driver

SYNOPSIS

```
STATUS pcConDrv (void)
```

DESCRIPTION This routine initializes the console driver, sets up interrupt vectors, and performs hardware initialization of the keybord and display.

RETURNS **OK**, or **ERROR** if the driver cannot be installed.

ERRNO Not Available

SEE ALSO **vxbPcConsole**

pentiumBtc( )

NAME **pentiumBtc( )** – execute atomic compare-and-exchange instruction to clear a bit

SYNOPSIS

```
STATUS pentiumBtc (pFlag)
    char * pFlag;                    /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with **TRUE**. If it is **TRUE**, it changes it to 0 and returns **OK**. If it is not **TRUE**, it returns **ERROR**. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS **OK** or **ERROR** if the specified flag is not **TRUE**

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumBts( )

NAME **pentiumBts( )** – execute atomic compare-and-exchange instruction to set a bit

SYNOPSIS

```
STATUS pentiumBts (pFlag)
    char * pFlag;                   /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with 0. If it is 0, it changes it to **TRUE** and returns **OK**. If it is not 0, it returns **ERROR**. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS **OK** or **ERROR** if the specified flag is not zero.

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumCr4Get( )

NAME **pentiumCr4Get( )** – get contents of CR4 register

SYNOPSIS

```
int pentiumCr4Get (void)
```

DESCRIPTION This routine gets the contents of the CR4 register. This routine is kept for the backward compatibility, and **vxCr4Get( )** should be used instead. The CR4 is introduced in the Pentium processor, thus this routine just returns in the pre Pentium generation processors.

RETURNS Contents of CR4 register.

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumCr4Set( )

NAME **pentiumCr4Set( )** – sets specified value to the CR4 register

SYNOPSIS

```
void pentiumCr4Set (cr4)
```

```
    int cr4;          /* value to write CR4 register */
```

DESCRIPTION This routine sets a specified value to the CR4 register. This routine is kept for the backward compatibility, and **vxCr4Set( )** should be used instead. The CR4 is introduced in the Pentium processor, thus this routine just returns in the pre Pentium generation processors.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumMcaEnable( )

NAME **pentiumMcaEnable( )** – enable/disable the MCA (Machine Check Architecture)

SYNOPSIS

```
void pentiumMcaEnable
    (
    BOOL enable  /* TRUE to enable, FALSE to disable the MCA */
    )
```

DESCRIPTION This routine enables/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.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumMcaShow( )

NAME **pentiumMcaShow( )** – show MCA (Machine Check Architecture) registers

SYNOPSIS

```
void pentiumMcaShow (void)
```

DESCRIPTION This routine shows Machine-Check global control registers and Error-Reporting register banks. Number of the Error-Reporting register banks is kept in a variable mcaBanks.

MCi_ADDR and MCi_MISC registers in the Error-Reporting register bank are showed if MCi_STATUS indicates that these registers are valid.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumShow**

pentiumMsrGet( )

NAME **pentiumMsrGet( )** – get the contents of the specified MSR (Model Specific Register)

SYNOPSIS

```
void pentiumMsrGet (addr, pData)
    int addr;                        /* MSR address */
    long long int * pData;           /* MSR data */
```

DESCRIPTION This routine gets the contents of the specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumMsrInit( )

NAME **pentiumMsrInit( )** – initialize all the MSRs (Model Specific Register)

SYNOPSIS

```
STATUS pentiumMsrInit (void)
```

DESCRIPTION This routine initializes all the MSRs in the processor. This routine works on either P5, P6 or P7 family processors.

RETURNS **OK**, or **ERROR** if RDMSR/WRMSR instructions are not supported.

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumMsrSet( )

NAME **pentiumMsrSet( )** – set a value to the specified MSR (Model Specific Registers)

SYNOPSIS

```
void pentiumMsrSet (addr, pData)
    int addr;                               /* MSR address */
    long long int * pData;          /* MSR data */
```

DESCRIPTION This routine sets a value to a specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumMsrShow( )

NAME **pentiumMsrShow( )** – show all the MSR (Model Specific Register)

SYNOPSIS

```
void pentiumMsrShow (void)
```

DESCRIPTION This routine shows all the MSRs in the Pentium and Pentium[234].

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumShow**

pentiumMtrrDisable( )

NAME

pentiumMtrrDisable( ) – disable MTRR (Memory Type Range Register)

SYNOPSIS

```
void pentiumMtrrDisable (void)
```

DESCRIPTION

This routine disables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

pentiumLib

pentiumMtrrEnable( )

NAME

pentiumMtrrEnable( ) – enable MTRR (Memory Type Range Register)

SYNOPSIS

```
void pentiumMtrrEnable (void)
```

DESCRIPTION

This routine enables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

pentiumLib

pentiumMtrrGet( )

NAME

pentiumMtrrGet( ) – get MTRRs to a specified MTRR table

SYNOPSIS

```
STATUS pentiumMtrrGet
    (
    MTRR * pMtrr  /* MTRR table */
    )
```

DESCRIPTION This routine gets MTRRs to a specified MTRR table with RDMSR instruction. The read MTRRs are CAP register, DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS **OK**, or **ERROR** if MTRR is being accessed.

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumMtrrSet( )

NAME **pentiumMtrrSet( )** – set MTRRs from specified MTRR table with WRMSR instruction.

SYNOPSIS

```
STATUS pentiumMtrrSet
    (
    MTRR * pMtrr  /* MTRR table */
    )
```

DESCRIPTION This routine sets MTRRs from specified MTRR table with WRMSR instruction. The written MTRRs are DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS **OK**, or **ERROR** if MTRR is enabled or being accessed.

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumP5PmcGet( )

NAME **pentiumP5PmcGet( )** – get the contents of P5 PMC0 and PMC1

SYNOPSIS

```
void pentiumP5PmcGet (pPmc0, pPmc1)
    long long int * pPmc0;            /* Performance Monitoring Counter 0 */
    long long int * pPmc1;            /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets the contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcGet0( )

NAME **pentiumP5PmcGet0( )** – get the contents of P5 PMC0

SYNOPSIS

```
void pentiumP5PmcGet0 (pPmc0)
    long long int * pPmc0;            /* Performance Monitoring Counter 0 */
```

DESCRIPTION This routine gets the contents of PMC0 (Performance Monitoring Counter 0). The parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcGet1( )

NAME **pentiumP5PmcGet1( )** – get the contents of P5 PMC1

SYNOPSIS

```
void pentiumP5PmcGet1 (pPmc1)
    long long int * pPmc1;            /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcReset( )

NAME **pentiumP5PmcReset( )** – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumP5PmcReset (void)
```

DESCRIPTION This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcReset0( )

NAME **pentiumP5PmcReset0( )** – reset PMC0

SYNOPSIS

```
void pentiumP5PmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcReset1( )

NAME **pentiumP5PmcReset1( )** – reset PMC1

SYNOPSIS

```
void pentiumP5PmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcStart0( )

NAME **pentiumP5PmcStart0( )** – start PMC0

SYNOPSIS

```
STATUS pentiumP5PmcStart0 (pmc0Cesr)
      int pmc0Cesr;           /* PMC0 control and event select */
```

DESCRIPTION This routine starts PMC0 (Performance Monitoring Counter 0) by writing specified PMC0 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC0 is already started.

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcStart1( )

NAME **pentiumP5PmcStart1( )** – start PMC1

SYNOPSIS

```
STATUS pentiumP5PmcStart1 (pmc1Cesr)
      int pmc1Cesr;           /* PMC1 control and event select */
```

DESCRIPTION This routine starts PMC1 (Performance Monitoring Counter 0) by writing specified PMC1 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC1 is already started.

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcStop( )

NAME **pentiumP5PmcStop( )** – stop both P5 PMC0 and PMC1

SYNOPSIS

```
void pentiumP5PmcStop (void)
```

DESCRIPTION This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcStop0( )

NAME **pentiumP5PmcStop0( )** – stop P5 PMC0

SYNOPSIS

```
void pentiumP5PmcStop0 (void)
```

DESCRIPTION This routine stops only PMC0 (Performance Monitoring Counter 0) by clearing the PMC0 bits of Control and Event Select Register.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP5PmcStop1( )

NAME **pentiumP5PmcStop1( )** – stop P5 PMC1

SYNOPSIS

```
void pentiumP5PmcStop1 (void)
```

DESCRIPTION This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the PMC1 bits of Control and Event Select Register.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcGet( )

NAME **pentiumP6PmcGet( )** – get the contents of PMC0 and PMC1

SYNOPSIS

```
void pentiumP6PmcGet (pPmc0, pPmc1)
    long long int * pPmc0;           /* Performance Monitoring Counter 0 */
    long long int * pPmc1;           /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets the contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcGet0( )

NAME **pentiumP6PmcGet0( )** – get the contents of PMC0

SYNOPSIS

```
void pentiumP6PmcGet0 (pPmc0)
    long long int * pPmc0;           /* Performance Monitoring Counter 0 */
```

DESCRIPTION This routine gets the contents of PMC0 (Performance Monitoring Counter 0). The parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcGet1( )

NAME **pentiumP6PmcGet1( )** – get the contents of PMC1

SYNOPSIS

```
void pentiumP6PmcGet1 (pPmc1)
    long long int * pPmc1;           /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcReset( )

NAME **pentiumP6PmcReset( )** – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumP6PmcReset (void)
```

DESCRIPTION This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcReset0( )

NAME **pentiumP6PmcReset0( )** – reset PMC0

SYNOPSIS

```
void pentiumP6PmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcReset1( )

NAME **pentiumP6PmcReset1( )** – reset PMC1

SYNOPSIS

```
void pentiumP6PmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcStart( )

NAME **pentiumP6PmcStart( )** – start both PMC0 and PMC1

SYNOPSIS

```
STATUS pentiumP6PmcStart (pmcEvtSel0, pmcEvtSel1)
    int pmcEvtSel0;          /* Performance Event Select Register 0 */
    int pmcEvtSel1;          /* Performance Event Select Register 1 */
```

DESCRIPTION This routine starts both PMC0 (Performance Monitoring Counter 0) and PMC1 by writing specified events to Performance Event Select Registers. The first parameter is a content of Performance Event Select Register 0, and the second parameter is for the Performance Event Select Register 1.

RETURNS **OK** or **ERROR** if PMC is already started.

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcStop( )

NAME **pentiumP6PmcStop( )** – stop both PMC0 and PMC1

SYNOPSIS
```
void pentiumP6PmcStop (void)
```

DESCRIPTION This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumP6PmcStop1( )

NAME **pentiumP6PmcStop1( )** – stop PMC1

SYNOPSIS
```
void pentiumP6PmcStop1 (void)
```

DESCRIPTION This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the Performance Event Select Register 1. Note, clearing the Performance Event Select Register 0 stops both counters, PMC0 and PMC1.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumPmcGet( )

NAME **pentiumPmcGet( )** – get the contents of PMC0 and PMC1

SYNOPSIS
```
void pentiumPmcGet (pPmc0, pPmc1)
    long long int * pPmc0;            /* Performance Monitoring Counter 0 */
    long long int * pPmc1;            /* Performance Monitoring Counter 1 */
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcGet0( )

NAME **pentiumPmcGet0( )** – get the contents of PMC0

SYNOPSIS

```
void pentiumPmcGet0 (pPmc0)
    long long int * pPmc0;            /* Performance Monitoring Counter 0 */
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcGet1( )

NAME **pentiumPmcGet1( )** – get the contents of PMC1

SYNOPSIS

```
void pentiumPmcGet1 (pPmc1)
    long long int * pPmc1;            /* Performance Monitoring Counter 1 */
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcReset( )

NAME **pentiumPmcReset( )** – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcReset (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcReset0( )

NAME **pentiumPmcReset0( )** – reset PMC0

SYNOPSIS

```
void pentiumPmcReset0 (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcReset1( )

NAME **pentiumPmcReset1( )** – reset PMC1

SYNOPSIS

```
void pentiumPmcReset1 (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcShow( )

NAME **pentiumPmcShow( )** – show PMCs (Performance Monitoring Counters)

SYNOPSIS

```
void pentiumPmcShow
    (
    BOOL zap  /* 1: reset PMC0 and PMC1 */
    )
```

DESCRIPTION This routine shows Performance Monitoring Counter 0 and 1. Monitored events are selected by Performance Event Select Registers in in pentiumPmcStart (). These counters are cleared to 0 if the parameter "zap" is **TRUE**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumShow**

pentiumPmcStart( )

NAME **pentiumPmcStart( )** – start both PMC0 and PMC1

SYNOPSIS

```
STATUS pentiumPmcStart (pmcEvtSel0, pmcEvtSel1)
    int pmcEvtSel0;           /* Performance Event Select Register 0 */
    int pmcEvtSel1;           /* Performance Event Select Register 1 */
```

DESCRIPTION none

RETURNS **OK** or **ERROR** if PMC is already started.

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcStart0( )

NAME
pentiumPmcStart0( ) – start PMC0

SYNOPSIS
```
STATUS pentiumPmcStart0 (pmcEvtSel0)
      int pmcEvtSel0;            /* PMC0 control and event select */
```

DESCRIPTION
none

RETURNS
OK or **ERROR** if PMC is already started.

ERRNO
Not Available

SEE ALSO
pentiumLib

pentiumPmcStart1( )

NAME
pentiumPmcStart1( ) – start PMC1

SYNOPSIS
```
STATUS pentiumPmcStart1 (pmcEvtSel1)
      int pmcEvtSel1;            /* PMC1 control and event select */
```

DESCRIPTION
none

RETURNS
OK or **ERROR** if PMC1 is already started.

ERRNO
Not Available

SEE ALSO
pentiumLib

pentiumPmcStop( )

NAME
pentiumPmcStop( ) – stop both PMC0 and PMC1

SYNOPSIS
```
void pentiumPmcStop (void)
```

DESCRIPTION
none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcStop0( )

NAME **pentiumPmcStop0( )** – stop PMC0

SYNOPSIS

```
void pentiumPmcStop0 (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumPmcStop1( )

NAME **pentiumPmcStop1( )** – stop PMC1

SYNOPSIS

```
void pentiumPmcStop1 (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumLib**

pentiumSerialize( )

NAME **pentiumSerialize( )** – execute a serializing instruction CPUID

SYNOPSIS
```
void pentiumSerialize (void)
```

DESCRIPTION This routine executes a serializing instruction CPUID. Serialization means that all modifications to flags, registers, and memory by previous instructions are completed before the next instruction is fetched and executed and all buffered writes have drained to memory.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumTlbFlush( )

NAME **pentiumTlbFlush( )** – flush TLBs (Translation Lookaside Buffers)

SYNOPSIS
```
void pentiumTlbFlush (void)
```

DESCRIPTION This routine flushes TLBs by loading the CR3 register. All of the TLBs are automatically invalidated any time the CR3 register is loaded. The page global enable (PGE) flag in register CR4 and the global flag in a page-directory or page-table entry can be used to frequently used pages from being automatically invalidated in the TLBs on a load of CR3 register. The only way to deterministically invalidate global page entries is to clear the PGE flag and then invalidate the TLBs.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

pentiumTscGet32( )

NAME
pentiumTscGet32( ) – get the lower half of the 64Bit TSC (Timestamp Counter)

SYNOPSIS
```
UINT32 pentiumTscGet32 (void)
```

DESCRIPTION
This routine gets a lower half of the 64Bit TSC by RDTSC instruction. RDTSC instruction saves the lower 32Bit in EAX register, so this routine simply returns after executing RDTSC instruction.

RETURNS
Lower half of the 64Bit TSC (Timestamp Counter)

ERRNO
Not Available

SEE ALSO
pentiumALib

pentiumTscGet64( )

NAME
pentiumTscGet64( ) – get 64Bit TSC (Timestamp Counter)

SYNOPSIS
```
void pentiumTscGet64 (pTsc)
    long long int * pTsc;            /* Timestamp Counter */
```

DESCRIPTION
This routine gets 64Bit TSC by RDTSC instruction. Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
pentiumALib

pentiumTscReset( )

NAME
pentiumTscReset( ) – reset the TSC (Timestamp Counter)

SYNOPSIS
```
void pentiumTscReset (void)
```

DESCRIPTION This routine resets the TSC by writing zero to the TSC with WRMSR instruction.

RETURNS N/A

ERRNO Not Available

SEE ALSO **pentiumALib**

period( )

NAME **period( )** – spawn a task to call a function periodically

SYNOPSIS

```
int period
    (
    int     secs,  /* period in seconds           */
    FUNCPTR func,  /* function to call repeatedly */
    int     arg1,  /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command spawns a task that repeatedly calls a specified function, with up to eight of its arguments, delaying the specified number of seconds between calls.

For example, to have **i( )** display task information every 5 seconds, just type:

```
    -> period 5, i
```

NOTE The task is spawned using the **sp( )** routine. See the description of **sp( )** for details about priority, options, stack size, and task ID.

RETURNS A task ID, or **ERROR** if the task cannot be spawned.

ERRNO **sp( )** errnos.

SEE ALSO **usrLib**, **periodRun( )**, **sp( )**, the VxWorks programmer guides.

periodRun( )

NAME **periodRun( )** – call a function periodically

SYNOPSIS

```
void periodRun
    (
    int     secs,  /* no. of seconds to delay between calls */
    FUNCPTR func,  /* function to call repeatedly */
    int     arg1,  /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command repeatedly calls a specified function, with up to eight of its arguments, delaying the specified number of seconds between calls.

Normally, this routine is called only by **period( )**, which spawns it as a task.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **period( )**, the VxWorks programmer guides.

philDemo( )

NAME **philDemo( )** – entry point for VxWorks/SMP Dijkstra's dining philosophers demo

SYNOPSIS

```
int philDemo (int arg)
```

DESCRIPTION This routine is the entry point for the VxWorks/SMP Dijkstra's dining philosophers demo. Specifying a non-0 value for *arg* disables the usage of ANSI escape sequences in the console output.

RETURNS **OK** always.

ERRNO N/A

SEE ALSO **phil**

pipeDevCreate( )

NAME

pipeDevCreate( ) – create a pipe device

SYNOPSIS

```
STATUS pipeDevCreate
    (
    const char* name,       /* name of pipe to be created      */
    int         nMessages,  /* max. number of messages in pipe */
    int         nBytes      /* size of each message            */
    )
```

DESCRIPTION

This routine creates a pipe device. It cannot be called from an interrupt service routine. It allocates memory for the necessary structures and initializes the device. The pipe device will have a maximum of *nMessages* messages of up to *nBytes* each in the pipe at once. When the pipe is full, a task attempting to write to the pipe will be suspended until a message has been read. Messages are lost if written to a full pipe at interrupt level.

RETURNS

OK, or **ERROR** if the call fails.

ERRNO

ENXIO
driver not initialized

S_intLib_NOT_ISR_CALLABLE
cannot be called from an ISR

EINVAL
invalid arguments

SEE ALSO

pipeDrv

pipeDevDelete( )

NAME

pipeDevDelete( ) – delete a pipe device

SYNOPSIS

```
STATUS pipeDevDelete
    (
    const char * name,  /* name of pipe to be deleted */
    BOOL         force  /* if TRUE, force pipe deletion */
    )
```

DESCRIPTION

This routine deletes a pipe device of a given name. The name must match that passed to **pipeDevCreate( )** else **ERROR** will be returned. This routine frees memory for the necessary structures and deletes the device. It cannot be called from an interrupt service routine.

A pipe device cannot be deleted until its number of open requests has been reduced to zero by an equal number of close requests and there are no tasks pending in its select list. If the optional force flag is asserted, the above restrictions are ignored, resulting in forced deletion of any select list and freeing of pipe resources.

CAVEAT Forced pipe deletion can have catastrophic results. Use only as a last resort.

RETURNS **OK**, or **ERROR** if the call fails.

ERRNO

S_intLib_NOT_ISR_CALLABLE
cannot be called from an ISR

ENXIO
driver not initialized

EMFILE
pipe still has open files

EBUSY
pipe is selected by at least one pending task

EINVAL
invalid arguments

ENODEV
no device found

SEE ALSO **pipeDrv**

pipeDrv( )

NAME **pipeDrv( )** – initialize the pipe driver

SYNOPSIS

```
STATUS pipeDrv (void)
```

DESCRIPTION This routine initializes and installs the driver. It must be called before any pipes are created. It is called automatically during initialization when VxWorks is configured with the **INCLUDE_PIPES** component.

RETURNS **OK**, or **ERROR** if the driver installation fails.

ERRNO

S_iosLib_DRIVER_GLUT (**ENOMEM**)
No memory available for data structures.

SEE ALSO **pipeDrv**

pmFreeSpace( )

NAME

pmFreeSpace( ) – returns the amount of free space left in the PM arena

SYNOPSIS

```
int pmFreeSpace
    (
    PM_ARENA_DEF arena  /* arena definition function    */
    )
```

DESCRIPTION

This function returns the amount of useable free space remaining in the PM arena. Clients of **pmLib** may request any amount up to this value.

RETURNS

the amount of free space in the arena (in bytes), or **ERROR** if **pmLib** has not been initialized

ERRNO

Not Available

SEE ALSO

pmLib

pmInvalidate( )

NAME

pmInvalidate( ) – invalidates the entire PM arena

SYNOPSIS

```
STATUS pmInvalidate
    (
    PM_ARENA_DEF arena  /* arena definition function    */
    )
```

DESCRIPTION

Warning: THIS ROUTINE WILL RENDER THE ENTIRE PM ARENA INVALID!

This function should be used with utmost care. It will invalidate the entire PM arena, effectively wiping out all regions and making their contents inaccessible.

It should only be used when there is a need to wipe the entire PM arena, typically during development.

RETURNS

OK or **ERROR**

ERRNO

Not Available

SEE ALSO

pmLib

pmRegionAddr( )

NAME

pmRegionAddr( ) – returns the address of a persistent heap region

SYNOPSIS

```
void *pmRegionAddr
    (
    PM_ARENA_DEF arena,  /* arena definition function    */
    int          region  /* region number number */
    )
```

DESCRIPTION

This function returns a pointer to the virtual address of the start of the data area of a persistent heap region.

RETURNS

a pointer to the region's data area or **NULL** if the region or arena is invalid

ERRNO

Not Available

SEE ALSO

pmLib

pmRegionClose( )

NAME

pmRegionClose( ) – closes a region making it inaccessible to clients

SYNOPSIS

```
STATUS pmRegionClose
    (
    PM_ARENA_DEF arena,  /* arena definition function    */
    int          region  /* region identifier            */
    )
```

DESCRIPTION

This function makes the given region inaccessible to all clients of **pmLib**. It does not guarantee to return the memory to the arena's free space, since it may be non-contiguous with the remaining free space, but it will attempt to coalesce it if at all possible.

RETURNS

OK if the region was closed safely, or **ERROR** if not

ERRNO

Not Available

SEE ALSO

pmLib

pmRegionCreate( )

NAME

pmRegionCreate( ) – creates a persistent heap region

SYNOPSIS

```
int pmRegionCreate
    (
    PM_ARENA_DEF arena,  /* arena definition function    */
    const char * key,    /* short name for region        */
    unsigned int size,   /* the requested size           */
    int          mode    /* initial protection           */
    )
```

DESCRIPTION

This function creates a new region in the PM arena, of the given size, with the given key. The key must be unique -- if a region already exists with the same name, it is considered an error, and this function will fail by returning **ERROR**.

If the requested size is not a multiple of the page size, it will be rounded up to the next multiple of the page size.

RETURNS

a positive integer identifying the region, or **ERROR** if it could not be created, or an existing region has the same name

ERRNO

Not Available

SEE ALSO

pmLib

pmRegionOpen( )

NAME

pmRegionOpen( ) – opens an existing persistent heap region

SYNOPSIS

```
int pmRegionOpen
    (
    PM_ARENA_DEF arena,  /* arena definition function    */
    const char * key     /* short name for region        */
    )
```

DESCRIPTION

This function opens an existing region in the PM arena. It looks for a region with a name matching the supplied key. If one is found, it will return its region identifier. If no such region is found, it returns **ERROR**.

RETURNS

a positive integer identifying the region, or **ERROR** if it could not be located

ERRNO

Not Available

SEE ALSO **pmLib**

pmRegionProtect( )

NAME **pmRegionProtect( )** – makes a PM region read-only

SYNOPSIS

```
STATUS pmRegionProtect
    (
    PM_ARENA_DEF arena,   /* arena definition function    */
    int          region,  /* the region identifier        */
    int          mode     /* PM_PROT_XXX value            */
    )
```

DESCRIPTION This function only alters the protection state of the region if the containing arena is set up for RDONLY mode, i.e. it prefers to be immutable most of the time. If the arena was initialised in RDWR mode, then it will remain writeable always, and any attempt to set it (or one of its regions) into RDONLY mode is an error.

RETURNS **OK** or **ERROR** if the region or arena is invalid

ERRNO Not Available

SEE ALSO **pmLib**

pmRegionSize( )

NAME **pmRegionSize( )** – return the size of a persistent heap region

SYNOPSIS

```
int pmRegionSize
    (
    PM_ARENA_DEF arena,  /* arena definition function     */
    int          region  /* persistent heap region number */
    )
```

DESCRIPTION This function returns the size of a region within the persistent heap.

RETURNS the size of the region's data area or **ERROR** if the region or arena is invalid

ERRNO Not Available

SEE ALSO **pmLib**

pmShow( )

NAME **pmShow( )** – shows the created persistent heap segments

SYNOPSIS

```
int pmShow
    (
    PM_ARENA_DEF arena  /* arena definition function    */
    )
```

DESCRIPTION This function displays the allocated persistent heaps and their headers.

RETURNS **OK** normally, or **ERROR** if the PM library is not initialised

ERRNO Not Available

SEE ALSO **pmLib**

pmValidate( )

NAME **pmValidate( )** – validates a PM arena

SYNOPSIS

```
STATUS pmValidate
    (
    PM_ARENA_DEF arena  /* arena definition function    */
    )
```

DESCRIPTION This function tests the validity or otherwise of a PM arena.

RETURNS **OK** if the arena is valid, or **ERROR** if it is corrupt or does not appear to be a PM arena

ERRNO Not Available

SEE ALSO **pmLib**

poolBlockAdd( )

NAME

poolBlockAdd( ) – add an item block to the pool

SYNOPSIS

```
ULONG poolBlockAdd
    (
    POOL_ID poolId,  /* ID of pool to delete */
    void *  pBlock,  /* base address of block to add */
    ULONG   size     /* size of block to add */
    )
```

DESCRIPTION

This routine adds an item block to the pool using memory provided by the user. The memory provided must be sufficient for at least one properly aligned item.

RETURNS

number of items added, or 0 in case of error

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

S_poolLib_INVALID_BLK_ADDR
pBlock parameter is **NULL**.

S_poolLib_BLOCK_TOO_SMALL
size insufficient for at least one item.

SEE ALSO

poolLib, **poolCreate( )**

poolCreate( )

NAME

poolCreate( ) – create a pool

SYNOPSIS

```
POOL_ID poolCreate
    (
    const char * pName,      /* optional name to assign to pool */
    ULONG        itmSize,    /* size in bytes of a pool item (must be > 0) */
    ULONG        alignment,  /* alignment of a pool item */
                             /* (must be power of 2, or 0) */
    ULONG        initCnt,    /* initial number of items to put in pool */
    ULONG        incrCnt,    /* min no of items to add to pool dynamically */
                             /* (if 0, no pool expansion is done) */
    PART_ID      partId,     /* memory partition ID */
    ULONG        options     /* initial options for pool */
    )
```

DESCRIPTION

This routine creates a pool by allocating an initial block of memory which is guarenteed to contain at least *initCnt* items. The pool will hold items of the specified size and alignment only. The alignment defaults to the architecture specific allocation alignment size, and it must be a power of two value. As items are allocated from the pool, the initial block may be emptied. When a block is emptied and more items are requested, another block of memory is dynamically allocated which is guarenteed to contain *incrCnt* items. If *incrCnt* is zero, no automatic pool expansion is done.

The partition ID parameter can be used to request all item blocks being allocated from a specific memory partition. If this parameter is **NULL**, the item blocks are allocated from the system memory partition.

POOL OPTIONS

The options parameter can be used to set the following properties of the pool. Options cannot be changed after the pool has been created. The following options are supported:

Option	Description
POOL_THREAD_SAFE	Pool operations are protected with mutex semaphore
POOL_CHECK_ITEM	Items returned to the pool are verified to be valid

RETURNS

ID of pool or **NULL** if any zero count or size or insufficient memory.

ERRNO

S_poolLib_ARG_NOT_VALID
one or more invalid input arguments.

SEE ALSO

poolLib, **poolDelete( )**

poolDelete( )

NAME

poolDelete( ) – delete a pool

SYNOPSIS

```
STATUS poolDelete
    (
    POOL_ID poolId,  /* ID of pool to delete */
    BOOL    force    /* force deletion if there are items in use */
    )
```

DESCRIPTION

This routine deletes a specified pool and all item blocks allocated for it. Memory provided by the user using **poolBlockAdd( )** are not freed.

If the pool is still in use (i.e. not all items have been returned to the pool) deletion can be forced with the *force* parameter set to **TRUE**.

RETURNS

OK or **ERROR** if bad pool ID or pool in use.

ERRNO **S_poolLib_INVALID_POOL_ID**
not a valid pool ID.

S_poolLib_POOL_IN_USE
can't delete a pool still in use.

SEE ALSO **poolLib**, **poolCreate( )**

poolFreeCount( )

NAME **poolFreeCount( )** – return number of free items in pool

SYNOPSIS

```
ULONG poolFreeCount
    (
    POOL_ID poolId  /* ID of pool */
    )
```

DESCRIPTION This routine returns the number of free items in the specified pool.

RETURNS number of items, or zero if invalid pool ID.

ERRNO **S_poolLib_INVALID_POOL_ID**
not a valid pool ID.

SEE ALSO **poolLib**, **poolTotalCount( )**

poolIncrementGet( )

NAME **poolIncrementGet( )** – get the increment value used to grow the pool

SYNOPSIS

```
ULONG poolIncrementGet
    (
    POOL_ID poolId  /* ID of pool */
    )
```

DESCRIPTION This routine can be used to get the increment value used to grow the pool. The increment specifies how many new items are added to the pool when there are no free items left in the pool.

RETURNS increment value, or zero if invalid pool ID.

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

SEE ALSO

poolLib, **poolIncrementSet( )**

poolIncrementSet( )

NAME

poolIncrementSet( ) – set the increment value used to grow the pool

SYNOPSIS

```
STATUS poolIncrementSet
    (
    POOL_ID poolId,  /* ID of pool */
    ULONG   incrCnt  /* new increment value */
    )
```

DESCRIPTION

This routine can be used to set the increment value used to grow the pool. The increment specifies how many new items are added to the pool when there are no free items left in the pool.

Setting the increment to zero disables automatic growth of the pool.

RETURNS

OK, or **ERROR** if poolId is invalid

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

SEE ALSO

poolLib, **poolIncrementGet( )**

poolItemGet( )

NAME

poolItemGet( ) – get next free item from pool and return a pointer to it

SYNOPSIS

```
void * poolItemGet
    (
    POOL_ID poolId  /* ID of pool from which to get item */
    )
```

DESCRIPTION

This routine gets the next free item from the specified pool and returns a pointer to it. If the current block of items is empty, the pool increment count is non-zero, and the routine is called from task context then a new block is allocated of the given incremental size and an item from the new block is returned.

In the kernel, this routine can be called from interrupt context if the pool was created without the **POOL_THREAD_SAFE** option. When called from ISR, the pool will not automatically grow and the routine fails if there are no free items in the pool.

RETURNS pointer to item, or **NULL** in case of error.

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

S_poolLib_STATIC_POOL_EMPTY
no more items available in static pool.

S_poolLib_INT_CTX_POOL_EMPTY
no more items in pool while called from ISR.

SEE ALSO **poolLib**, **poolItemReturn( )**

poolItemReturn( )

NAME **poolItemReturn( )** – return an item to the pool

SYNOPSIS

```
STATUS poolItemReturn
    (
    POOL_ID poolId,  /* ID of pool to which to return item */
    void *  pItem    /* pointer to item to return */
    )
```

DESCRIPTION This routine returns the specified item to the specified pool. To enable address verification on the item, the pool should be created with the **POOL_CHECK_ITEM** option. The verification can be an expensive operation, therefore the **POOL_CHECK_ITEM** option should be used when error detection is more important than deterministic behaviour of this routine.

In the kernel, this routine can be called from an ISR if the pool was created without the **POOL_THREAD_SAFE** option.

RETURNS **OK**, or **ERROR** in case of failure.

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

S_poolLib_NOT_POOL_ITEM
NULL pointer or item does not belong to pool.

S_poolLib_UNUSED_ITEM
item is already in pool free list.

SEE ALSO **poolLib**, **poolItemGet( )**

poolShow( )

NAME **poolShow( )** – display pool information

SYNOPSIS

```
void poolShow
    (
    POOL_ID poolId,  /* ID of pool from which to get item */
    ULONG   level    /* display info level */
    )
```

DESCRIPTION This show routine displays information about a pool. If level is 1, it also displays statistics about memory usage efficiency by the pool. Some count values and statistics typically change dynamically, so the displayed values represent a snapshot of the pool status at the time of querying.

If the pool ID passed to this routine is **NULL**, a summary of all pools managed by **poolLib** is displayed (up to 128 pools). The following is an example for a summary info:

EXAMPLE

```
-> poolShow

 NAME                    POOL ID     SIZE    TOTAL     FREE
-------------------- ---------- -------- -------- --------
fdEntries            0x02439ef0       80      450       44
sets                 0x02439d00       84       72        7
set_nodes            0x02439a60       12      288       31
mmuPgTables          0x02438f60     4096     1647        3
memEdrPool           0x02338d20       32   294913    26973
```

The following is an example for a detailed info for a specific pool, with info level 1:

EXAMPLE

```
-> poolShow 0x02438f60, 1

Pool         : mmuPgTables
Item Size    : 4096
Alignment    : 0x1000
Increment    : 8
Total items  : 1647
Free items   : 3
Options      : THREAD_SAFE
Blocks       : 2
```

```
  Overhead    : 204 bytes (0%)

      BLOCK ADDR    ITEMS     FREE
      ----------  --------  --------
      0x024ea000         8         3
      0x0243a000       175         0
```

If the pool ID passed to this routine is **NULL**, a summary of all pools managed by **poolLib** is displayed (up to 128 pools).

RETURNS N/A

ERRNO none

SEE ALSO **poolShow**, **poolLib**

poolTotalCount( )

NAME **poolTotalCount( )** – return total number of items in pool

SYNOPSIS

```
ULONG poolTotalCount
    (
    POOL_ID poolId  /* ID of pool */
    )
```

DESCRIPTION This routine returns the total number of items in the specified pool.

RETURNS number of items, or zero if invalid pool ID.

ERRNO **S_poolLib_INVALID_POOL_ID**
not a valid pool ID.

SEE ALSO **poolLib**, **poolFreeCount( )**

poolUnusedBlocksFree( )

NAME **poolUnusedBlocksFree( )** – free blocks that have all items unused

SYNOPSIS

```
STATUS poolUnusedBlocksFree
```

```
    (
    POOL_ID poolId  /* ID of pool to free blocks */
    )
```

DESCRIPTION

This routine allows reducing the memory used by a pool by freeing item blocks that have all items returned to the pool. Execution time of this routine is not deterministic as it depends on the number of free items and the number of blocks in the pool. In case of multi-thread safe pools (**POOL_THREAD_SAFE**), this routine also locks the pool for that time.

Blocks that were added using **poolBlockAdd( )** are not freed by this routine, even if all items have been returned; only blocks that were automatically allocated during creation or auto-growth from the pool's memory partition are freed.

RETURNS

OK, or **ERROR** in case of failure

ERRNO

S_poolLib_INVALID_POOL_ID
not a valid pool ID.

SEE ALSO

poolLib, **poolBlockAdd( )**, **poolCreate( )**

powf( )

NAME

powf( ) – compute the value of a number raised to a specified power (ANSI)

SYNOPSIS

```
float powf
    (
    float x,  /* operand  */
    float y   /* exponent */
    )
```

DESCRIPTION

This routine returns the value of *x* to the power of *y* in single precision.

RETURNS

The single-precision value of *x* to the power of *y*.

ERRNO

Not Available

SEE ALSO

mathALib

primesCompute( )

NAME

primesCompute( ) – entry point for the VxWorks SMP prime number computation demo

SYNOPSIS

```
STATUS primesCompute
    (
    unsigned int maxPrimeNum,
    unsigned int numTasks      /* if 0 then use display mode */
    )
```

DESCRIPTION

This routine is the entry point for the VxWorks SMP prime number computation demo.

This function will create *numTasks* computational tasks to compute prime numbers from 2 to *maxPrimeNum*. 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.

See the module description for more information.

RETURNS

ERROR if failed to allocate memory for the prime number candidate array or failed to spawn computational tasks. Otherwise **OK** is returned.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
Out of memory for creation of computational tasks or prime number candidate array

SEE ALSO

primesDemo

printErr( )

NAME

printErr( ) – write a formatted string to the standard error stream

SYNOPSIS

```
int printErr
    (
    const char * fmt,  /* format string to write */
    ...                /* optional arguments to format */
    )
```

DESCRIPTION

This routine writes a formatted string to standard error. Its function and syntax are otherwise identical to **printf( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

ERRNO Not Available

SEE ALSO **fioBaseLib**, **printf( )**

printErrno( )

NAME **printErrno( )** – print the definition of a specified error status value

SYNOPSIS

```
void printErrno
    (
    int errNo  /* status code whose name is to be printed */
    )
```

DESCRIPTION This command displays the error-status string, corresponding to a specified error-status value. It is only useful if the error-status symbol table has been built and included in the system. If *errNo* is zero, then the current task status is used by calling **errnoGet( )**.

This facility is described in **errnoLib**.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **errnoLib**, **errnoGet( )**, the VxWorks programmer guides.

printLogo( )

NAME **printLogo( )** – print the VxWorks logo

SYNOPSIS

```
void printLogo (void)
```

DESCRIPTION This command displays the VxWorks banner seen at boot time. It also displays the VxWorks version number and kernel version number.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, the VxWorks programmer guides.

printf( )

NAME **printf( )** – write a formatted string to the standard output stream (ANSI)

SYNOPSIS

```
int printf
    (
    const char * fmt,  /* format string to write */
    ...                /* optional arguments to format string */
    )
```

DESCRIPTION This routine writes output to standard output under control of the string *fmt*. The string *fmt* contains ordinary characters, which are written unchanged, plus conversion specifications, which cause the arguments that follow *fmt* to be converted and printed as part of the formatted string.

The number of arguments for the format is arbitrary, but they must correspond to the conversion specifications in *fmt*. If there are insufficient arguments, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The routine returns when the end of the format string is encountered.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not **%**) that are copied unchanged to the output stream; and conversion specification, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (*****) (described later) or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in the **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk (*****) (described later) or by an optional decimal integer; if only the period is

specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

- An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument. An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **long int** or **unsigned long int** argument; or an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument. An optional **ll** (ell-ell) specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **long long int** or `unsigned long long int' argument; or an optional **ll** specifying that a following **n** conversion specifier applies to a pointer to a **long long int** argument. If a **h**, **l** or **ll** appears with any other conversion specifier, the behavior is undefined.
- WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** argument. However, the current release of the VxWorks libraries does not support **long double** data; using the optional **L** gives unpredictable results.
- A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, can be indicated by an asterisk (*). In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, should appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

-
: The result of the conversion will be left-justified within the field. (it will be right-justified if this flag is not specified.)

+
: The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space
: If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the **space** and **+** flags both appear, the **space** flag will be ignored.

#
: The result is to be converted to an "alternate form." For **o** conversion it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a non-zero result will have "0x" (or "0X") prefixed to it. For **e**, **E**, **f**, **g**, and **g** conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if

no digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result. For other conversions, the behavior is undefined.

0
: For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

d, **i**
: The **int** argument is converted to signed decimal in the style **[-]dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o, **u**, **x**, **X**
: The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style **dddd**; the letters abcdef are used for **x** conversion and the letters ABCDEF for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f
: The **double** argument is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e, **E**
: The **double** argument is converted in the style **[-]d.ddde+/-dd**, where there is one digit before the decimal-point character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g, **G**
: The **double** argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted;

style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

c
The **int** argument is converted to an **unsigned char**, and the resulting character is written.

s
The argument should be a pointer to an array of character type. Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array will contain a null character.

p
The argument should be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in hexadecimal representation (prefixed with "0x").

n
The argument should be a pointer to an integer into which the number of characters written to the output stream so far by this call to **fprintf( )** is written. No argument is converted.

%
A **%** is written. No argument is converted. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using **s** conversion, or a pointer using **p** conversion), the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted.

RETURNS
The number of characters written, or a negative value if an output error occurs.

ERRNO
Not Available

SEE ALSO
fioBaseLib, **fprintf( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output (***stdio.h***)*

proofUtf8( )

NAME

proofUtf8( ) – Determine if a string represents a valid UTF-8 character

SYNOPSIS

```
int proofUtf8
    (
    const unsigned char * utf8,
    const int             length
    )
```

DESCRIPTION

This routine checks a string to determine if it contains a valid UTF-8 encoded character, including **\0**.

RETURNS

If positive, the number of encoded bytes used to represent the Unicode character. If non-positive, **UC_FORMAT** indicates that the string is in an invalid format, and **UC_NOSRC** indicates that the string contains insufficient characters to represent a valid encoding, given the value of the first character.

ERRNO

Not Available

SEE ALSO

utfLib

proofUtf8String( )

NAME

proofUtf8String( ) – determine if a string is valid UTF-8

SYNOPSIS

```
int proofUtf8String
    (
    const unsigned char * utf8
    )
```

DESCRIPTION

This routine determines if a **NULL** terminated string is valid UTF-8.

RETURNS

If positive, the number of Unicode characters represented by a UTF-8 encoding. If non-positive, **UC_FORMAT** indicates that the string is of invalid format.

ERRNO

Not Available

SEE ALSO

utfLib

psr( )

NAME **psr( )** – return the contents of the processor status register (SimSolaris)

SYNOPSIS

```
int psr
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the processor status register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS The contents of the processor status register.

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

psrShow( )

NAME **psrShow( )** – display the meaning of a specified PSR value, symbolically (ARM)

SYNOPSIS

```
STATUS psrShow
    (
    UINT32 psrval  /* psr value to show */
    )
```

DESCRIPTION This routine displays the meaning of all fields in a specified PSR value, symbolically.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **dbgArchLib**

pthread_attr_destroy( )

NAME **pthread_attr_destroy( )** – destroy a thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_destroy
    (
    pthread_attr_t *pAttr  /* thread attributes */
    )
```

DESCRIPTION Destroy the thread attributes object *pAttr*. It should not be re-used until it has been reinitialized.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_init( )**

pthread_attr_getdetachstate( )

NAME **pthread_attr_getdetachstate( )** – get value of detachstate attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getdetachstate
    (
    const pthread_attr_t *pAttr,        /* thread attributes             */
    int                  *pDetachstate  /* current detach state (out)    */
    )
```

DESCRIPTION This routine returns the current detach state specified in the thread attributes object *pAttr*. The value is stored in the location pointed to by *pDetachstate*. Possible values for the detach state are: **PTHREAD_CREATE_DETACHED** and **PTHREAD_CREATE_JOINABLE**.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed or if *pDetachState* is **NULL**.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_setdetachstate( )**

pthread_attr_getinheritsched( )

NAME

pthread_attr_getinheritsched( ) – get current value if inheritsched attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getinheritsched
    (
    const pthread_attr_t *pAttr,          /* thread attributes object     */
    int                  *pInheritsched  /* inheritance mode (out)       */
    )
```

DESCRIPTION

This routine gets the scheduling inheritance value from the thread attributes object *pAttr*.

Possible values are:

PTHREAD_INHERIT_SCHED
Inherit scheduling parameters from parent thread.

PTHREAD_EXPLICIT_SCHED
Use explicitly provided scheduling parameters (i.e. those specified in the thread attributes object).

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO

N/A

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( ) pthread_attr_setinheritsched( )**

pthread_attr_getname( )

NAME

pthread_attr_getname( ) – get name of thread attribute object

SYNOPSIS

```
int pthread_attr_getname
    (
    pthread_attr_t *pAttr,
    char           **name
    )
```

DESCRIPTION

This routine gets the name in the specified thread attributes object, *pAttr*.

RETURNS

zero on success, **EINVAL** if an invalid thread attribute is passed or if *name* is **NULL**.

ERRNO

None.

SEE ALSO **pthreadLib**, **pthread_attr_setname( )**

pthread_attr_getopt( )

NAME **pthread_attr_getopt( )** – get options from thread attribute object

SYNOPSIS

```
int pthread_attr_getopt
    (
    pthread_attr_t * pAttr,
    int *            pOptions
    )
```

DESCRIPTION This non-POSIX routine gets options from the specified thread attributes object, *pAttr*. To see the options actually applied to the VxWorks task under thread, use **taskOptionsGet( )**.

This routine expects the *pOptions* parameter to be a valid storage space.

See *taskLib.h* for definitions of task options.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed or if *pOptions* is **NULL**.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_setopt( )**, **taskOptionsGet( )**

pthread_attr_getschedparam( )

NAME **pthread_attr_getschedparam( )** – get value of schedparam attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getschedparam
    (
    const pthread_attr_t *pAttr,  /* thread attributes            */
    struct sched_param   *pParam  /* current parameters (out)     */
    )
```

DESCRIPTION Return, via the pointer *pParam*, the current scheduling parameters from the thread attributes object *pAttr*.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_setschedparam( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_attr_getschedpolicy( )

NAME **pthread_attr_getschedpolicy( )** – get schedpolicy attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getschedpolicy
    (
    const pthread_attr_t *pAttr,   /* thread attributes    */
    int                  *pPolicy  /* current policy (out) */
    )
```

DESCRIPTION This routine returns, via the pointer *pPolicy*, the current scheduling policy in the thread attributes object specified by *pAttr*. Possible values for VxWorks systems are **SCHED_RR**, **SCHED_FIFO** and **SCHED_OTHER**.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_setschedpolicy( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_setscheduler( )**, **sched_getscheduler( )**

pthread_attr_getscope( )

NAME **pthread_attr_getscope( )** – get contention scope from thread attributes (POSIX)

SYNOPSIS

```
int pthread_attr_getscope
    (
    const pthread_attr_t *pAttr,            /* thread attributes object
*/
    int                  *pContentionScope  /* contention scope (out)
*/
    )
```

DESCRIPTION Reads the current contention scope setting from a thread attributes object. For VxWorks this is always **PTHREAD_SCOPE_SYSTEM**. If the thread attributes object is uninitialized then **EINVAL** will be returned. The contention scope is returned in the location pointed to by *pContentionScope*.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_setscope( )**

pthread_attr_getstackaddr( )

NAME **pthread_attr_getstackaddr( )** – get value of stackaddr attribute from thread attributes object (POSIX)

SYNOPSIS
```
int pthread_attr_getstackaddr
    (
    const pthread_attr_t *pAttr,        /* thread attributes             */
    void                 **ppStackaddr  /* current stack address (out)   */
    )
```

DESCRIPTION This routine returns the stack address from the thread attributes object *pAttr* in the location pointed to by *ppStackaddr*.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed or if *ppStackaddr* is **NULL**.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_getstacksize( )**, **pthread_attr_setstackaddr( )**

pthread_attr_getstacksize( )

NAME **pthread_attr_getstacksize( )** – get stack value of stacksize attribute from thread attributes object (POSIX)

SYNOPSIS
```
int pthread_attr_getstacksize
    (
    const pthread_attr_t *pAttr,      /* thread attributes             */
    size_t              *pStacksize  /* current stack size (out)     */
    )
```

DESCRIPTION This routine gets the current stack size from the thread attributes object *pAttr* and places it in the location pointed to by *pStacksize*.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed or if *pStackSize* is **NULL**.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_setstacksize( )**, **pthread_attr_getstackaddr( )**

pthread_attr_init( )

NAME **pthread_attr_init( )** – initialize thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_init
    (
    pthread_attr_t *pAttr  /* thread attributes */
    )
```

DESCRIPTION This routine initializes a thread attributes object. If *pAttr* is **NULL** then this function will return **EINVAL**.

The attributes that are set by default are as follows:

Stack Address
NULL - allow the system to allocate the stack.

Stack Size
0 - use the VxWorks **taskLib** default stack size.

Detach State
PTHREAD_CREATE_JOINABLE

Contention Scope
PTHREAD_SCOPE_SYSTEM

Scheduling Inheritance
PTHREAD_INHERIT_SCHED

Scheduling Policy
SCHED_OTHER (i.e. active VxWorks native scheduling policy).

Scheduling Priority
Use **pthreadLib** default priority

Note that the scheduling policy and priority values are only used if the scheduling inheritance mode is changed to **PTHREAD_EXPLICIT_SCHED** - see **pthread_attr_setinheritsched( )** for information.

Additionally, VxWorks-specific attributes are being set as follows:

Task Name
  NULL - the task name is automatically generated.

Task Options
  VX_FP_TASK

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_destroy( )**, **pthread_attr_getdetachstate( )**, **pthread_attr_getinheritsched( )**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_getscope( )**, **pthread_attr_getstackaddr( )**, **pthread_attr_getstacksize( )**, **pthread_attr_setdetachstate( )**, **pthread_attr_setinheritsched( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**, **pthread_attr_setscope( )**, **pthread_attr_setstackaddr( )**, **pthread_attr_setstacksize( )**, **pthread_attr_setname( )** (VxWorks extension), **pthread_attr_setopt( )** (VxWorks extension)

pthread_attr_setdetachstate( )

NAME **pthread_attr_setdetachstate( )** – set detachstate attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setdetachstate
    (
    pthread_attr_t *pAttr,      /* thread attributes    */
    int            detachstate  /* new detach state     */
    )
```

DESCRIPTION This routine sets the detach state in the thread attributes object *pAttr*. The new detach state specified by *detachstate* must be one of **PTHREAD_CREATE_DETACHED** or **PTHREAD_CREATE_JOINABLE**. Any other values will cause an error to be returned (**EINVAL**).

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getdetachstate( )**, **pthread_attr_init( )**

pthread_attr_setinheritsched( )

NAME **pthread_attr_setinheritsched( )** – set inheritsched attribute in thread attribute object (POSIX)

SYNOPSIS

```
int pthread_attr_setinheritsched
    (
    pthread_attr_t *pAttr,       /* thread attributes object     */
    int            inheritsched  /* inheritance mode             */
    )
```

DESCRIPTION This routine sets the scheduling inheritance to be used when creating a thread with the thread attributes object specified by *pAttr*.

Possible values are:

PTHREAD_INHERIT_SCHED
Inherit scheduling parameters from parent thread.

PTHREAD_EXPLICIT_SCHED
Use explicitly provided scheduling parameters (i.e. those specified in the thread attributes object).

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getinheritsched( )**, **pthread_attr_init( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**

pthread_attr_setname( )

NAME **pthread_attr_setname( )** – set name in thread attribute object

SYNOPSIS

```
int pthread_attr_setname
    (
    pthread_attr_t *pAttr,
    char           *name
    )
```

DESCRIPTION This routine sets the name in the specified thread attributes object, *pAttr*.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_getname( )**

pthread_attr_setopt( )

NAME **pthread_attr_setopt( )** – set options in thread attribute object

SYNOPSIS

```
int pthread_attr_setopt
    (
    pthread_attr_t * pAttr,
    int              options
    )
```

DESCRIPTION This non-POSIX routine sets options in the specified thread attributes object, *pAttr*. This allows for specifying a non-default set of options for the VxWorks task acting as a thread. Additional options may be applied to the task once the thread has been created via the **taskOptionsSet( )** API.

Note that the task options provided through this routine will supersede the default options otherwise applied at thread creation.

See *taskLib.h* for definitions of valid task options.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_getopt( )**, **taskOptionsSet( )**

pthread_attr_setschedparam( )

NAME **pthread_attr_setschedparam( )** – set schedparam attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setschedparam
```

```
    (
    pthread_attr_t           *pAttr,  /* thread attributes    */
    const struct sched_param *pParam  /* new parameters       */
    )
```

DESCRIPTION Set the scheduling parameters in the thread attributes object *pAttr*. The scheduling parameters are essentially the thread's priority. Note that the **PTHREAD_EXPLICIT_SCHED** mode must be set (see **pthread_attr_setinheritsched( )** for information) for the priority to take effect.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getschedparam( )**, **pthread_attr_init( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **pthread_attr_setinheritsched( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_attr_setschedpolicy( )

NAME **pthread_attr_setschedpolicy( )** – set schedpolicy attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setschedpolicy
    (
    pthread_attr_t *pAttr,  /* thread attributes    */
    int            policy   /* new policy           */
    )
```

DESCRIPTION Select the thread scheduling policy. The default scheduling policy is to inherit the current system setting. Unlike the POSIX model, scheduling policies under VxWorks are global. If a scheduling policy is being set explicitly, the **PTHREAD_EXPLICIT_SCHED** mode must be set (see **pthread_attr_setinheritsched( )** for information), and the selected scheduling policy must match the global scheduling policy in place at the time; failure to do so will result in **pthread_create( )** failing with the error **EPERM**.

POSIX defines the following policies:

SCHED_RR
Realtime, round-robin scheduling.

SCHED_FIFO
Realtime, first-in first-out scheduling.

SCHED_OTHER
Other, active VxWorks native scheduling policy.

Although the **SCHED_RR** and **SCHED_FIFO** policies can be set when the precaution described above is respected, using the **SCHED_OTHER** policy instead is always ensured to be successful.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getschedpolicy( )**, **pthread_attr_init( )**, **pthread_attr_setinheritsched( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_setscheduler( )**, **sched_getscheduler( )**

pthread_attr_setscope( )

NAME **pthread_attr_setscope( )** – set contention scope for thread attributes (POSIX)

SYNOPSIS

```
int pthread_attr_setscope
    (
    pthread_attr_t *pAttr,           /* thread attributes object     */
    int            contentionScope  /* new contention scope         */
    )
```

DESCRIPTION For VxWorks **PTHREAD_SCOPE_SYSTEM** is the only supported contention scope. If the **PTHREAD_SCOPE_PROCESS** value is passed to this function this will result in **ENOTSUP** being returned.

RETURNS On success zero; on failure the **EINVAL** or **ENOTSUP** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getscope( )**, **pthread_attr_init( )**

pthread_attr_setstackaddr( )

NAME **pthread_attr_setstackaddr( )** – set stackaddr attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setstackaddr
    (
    pthread_attr_t *pAttr,      /* thread attributes             */
    void           *pStackaddr  /* new stack address            */
    )
```

DESCRIPTION

This routine sets the stack address in the thread attributes object *pAttr* to be *pStackaddr*. On VxWorks this address must be the lowest address of the stack regardless of what the thread considers as the stack base or the stack end.

No alignment constraints are imposed by the pthread library so the thread's stack can be obtained via a simple call to **malloc( )** or **memPartAlloc( )**.

The memory area used a stack is not automatically freed when the thread exits. This operation cannot be done via the exiting thread's cleanup stack since the cleanup handler routines use the same stack as the thread. Therefore freeing the stack space must be done by the code which allocated the thread's stack once the thread's task no longer exists in the system.

The stack size is set using the routine **pthread_attr_setstacksize( )**. Note that failure to set the stack size when a stack address is provided will result in an **EINVAL** error status returned by **pthread_create( )**.

RETURNS

zero on success, **EINVAL** if an invalid thread attribute is passed.

ERRNO

None.

SEE ALSO

pthreadLib, **pthread_attr_getstacksize( )**, **pthread_attr_setstacksize( )**, **pthread_attr_init( )**

pthread_attr_setstacksize( )

NAME

pthread_attr_setstacksize( ) – set stacksize attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setstacksize
    (
    pthread_attr_t *pAttr,    /* thread attributes    */
    size_t         stacksize  /* new stack size       */
    )
```

DESCRIPTION

This routine sets the thread stack size (in bytes) in the specified thread attributes object, *pAttr*.

The stack address is set using the routine **pthread_attr_setstackaddr( )**. Note that failure to set the stack size when a stack address is provided will result in an **EINVAL** error status returned by **pthread_create( )**.

RETURNS

EINVAL if the stack size is lower than **PTHREAD_STACK_MIN** or if an invalid thread attribute is passed. Zero otherwise.

ERRNO
None.

SEE ALSO
pthreadLib, **pthread_attr_getstacksize( )**, **pthread_attr_setstackaddr( )**, **pthread_attr_init( )**, **pthread_create( )**

pthread_cancel( )

NAME
pthread_cancel( ) – cancel execution of a thread (POSIX)

SYNOPSIS
```
int pthread_cancel
    (
    pthread_t thread  /* thread to cancel */
    )
```

DESCRIPTION
This routine sends a cancellation request to the thread specified by *thread*. Depending on the settings of that thread, it may ignore the request, terminate immediately or defer termination until it reaches a cancellation point.

When the thread terminates it performs as if **pthread_exit( )** had been called with the exit status **PTHREAD_CANCELED**.

IMPLEMENTATION NOTE
In VxWorks, asynchronous thread cancellation is accomplished using a signal. The signal **SIGCNCL** has been reserved for this purpose. Applications should take care not to block or handle this signal.

RETURNS
On success zero; on failure the **ESRCH** error code.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_exit( )**, **pthread_setcancelstate( )**, **pthread_setcanceltype( )**, **pthread_testcancel( )**

pthread_cleanup_pop( )

NAME
pthread_cleanup_pop( ) – pop a cleanup routine off the top of the stack (POSIX)

SYNOPSIS
```
void pthread_cleanup_pop
```

```
    (
    int run  /* execute handler? */
    )
```

DESCRIPTION This routine removes the cleanup handler routine at the top of the cancellation cleanup stack of the calling thread and executes it if *run* is non-zero. The routine should have been added using the **pthread_cleanup_push( )** function.

Once the routine is removed from the stack it will no longer be called when the thread exits.

RETURNS N/A

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_cleanup_push( )**, **pthread_exit( )**

pthread_cleanup_push( )

NAME **pthread_cleanup_push( )** – pushes a routine onto the cleanup stack (POSIX)

SYNOPSIS

```
void pthread_cleanup_push
    (
    void (*routine)(void *),   /* cleanup routine      */
    void                *arg  /* argument             */
    )
```

DESCRIPTION This routine pushes the specified cancellation cleanup handler routine, *routine*, onto the cancellation cleanup stack of the calling thread. When a thread exits and its cancellation cleanup stack is not empty, the cleanup handlers are invoked with the argument *arg* in LIFO order from the cancellation cleanup stack.

RETURNS N/A

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_cleanup_pop( )**, **pthread_exit( )**

pthread_cond_broadcast( )

NAME

pthread_cond_broadcast( ) – unblock all threads waiting on a condition (POSIX)

SYNOPSIS

```
int pthread_cond_broadcast
    (
    pthread_cond_t *pCond
    )
```

DESCRIPTION

This function unblocks all threads blocked on the condition variable *pCond*. Nothing happens if no threads are waiting on the specified condition variable.

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO

N/A

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_destroy( )

NAME

pthread_cond_destroy( ) – destroy a condition variable (POSIX)

SYNOPSIS

```
int pthread_cond_destroy
    (
    pthread_cond_t *pCond  /* condition variable */
    )
```

DESCRIPTION

This routine destroys the condition variable pointed to by *pCond*. No threads can be waiting on the condition variable when this function is called. If there are threads waiting on the condition variable, then **pthread_cond_destroy( )** returns **EBUSY**.

RETURNS

On success zero; on failure a non-zero error code.

EINVAL

EBUSY

ERRNO

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_init( )

NAME **pthread_cond_init( )** – initialize condition variable (POSIX)

SYNOPSIS
```
int pthread_cond_init
    (
    pthread_cond_t     *pCond,  /* condition variable            */
    pthread_condattr_t *pAttr   /* condition variable attributes */
    )
```

DESCRIPTION This function initializes a condition variable. A condition variable is a synchronization device that allows threads to block until some predicate on shared data is satisfied. The basic operations on conditions are to signal the condition (when the predicate becomes true), and wait for the condition, blocking the thread until another thread signals the condition.

A condition variable must always be associated with a mutex to avoid a race condition between the wait and signal operations.

If *pAttr* is **NULL** then the default attributes are used as specified by POSIX; if *pAttr* is non-**NULL** then it is assumed to point to a condition attributes object initialized by **pthread_condattr_init( )**, and those are the attributes used to create the condition variable.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ERRNO

SEE ALSO **pthreadLib**, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_signal( )

NAME **pthread_cond_signal( )** – unblock a thread waiting on a condition (POSIX)

SYNOPSIS
```
int pthread_cond_signal
    (
    pthread_cond_t *pCond
    )
```

DESCRIPTION This routine unblocks one thread waiting on the specified condition variable *pCond*. If no threads are waiting on the condition variable then this routine does nothing; if more than one thread is waiting, then one will be released, but it is not specified which one.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_timedwait( )

NAME **pthread_cond_timedwait( )** – wait for a condition variable with a timeout (POSIX)

SYNOPSIS

```
int pthread_cond_timedwait
    (
    pthread_cond_t        *pCond,    /* condition variable   */
    pthread_mutex_t       *pMutex,   /* POSIX mutex          */
    const struct timespec *pAbstime  /* timeout time         */
    )
```

DESCRIPTION This function atomically releases the mutex *pMutex* and waits for another thread to signal the condition variable *pCond*. As with **pthread_cond_wait( )**, the mutex must be locked by the calling thread when **pthread_cond_timedwait( )** is called.

If the condition variable is signalled before the system time reaches the time specified by *pAbsTime*, then the mutex is re-acquired and the calling thread unblocked.

If the system time reaches or exceeds the time specified by *pAbsTime* before the condition is signalled, then the mutex is re-acquired, the thread unblocked and **ETIMEDOUT** returned.

If the calling thread gets cancelled while pending on the condition variable **pthread_cond_timedwait( )** will also re-acquire the mutex prior to executing the cancellation cleanup handlers (if any). The mutex will however be released prior to the thread exiting so that this mutex can be used by other threads.

NOTE The timeout is specified as an absolute value of the system clock in a *timespec* structure (see **clock_gettime( )** for more information). This is different from most VxWorks timeouts which are specified in ticks relative to the current time.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ETIMEDOUT

ERRNO

SEE ALSO **pthrbeadLib**, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_wait( )**

pthread_cond_wait( )

NAME **pthread_cond_wait( )** – wait for a condition variable (POSIX)

SYNOPSIS

```
int pthread_cond_wait
    (
    pthread_cond_t  *pCond,  /* condition variable   */
    pthread_mutex_t *pMutex  /* POSIX mutex          */
    )
```

DESCRIPTION This function atomically releases the mutex *pMutex* and waits for the condition variable *pCond* to be signalled by another thread. The mutex must be locked by the calling thread when **pthread_cond_wait( )** is called; if it is not then this function returns an error (**EINVAL**).

Before returning to the calling thread, **pthread_cond_wait( )** re-acquires the mutex.

If the calling thread gets cancelled while pending on the condition variable **pthread_cond_wait( )** will also re-acquire the mutex prior to executing the cancellation cleanup handlers (if any). The mutex will however be released prior to the thread exiting so that this mutex can be used by other threads.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**

pthread_condattr_destroy( )

NAME

pthread_condattr_destroy( ) – destroy a condition attributes object (POSIX)

SYNOPSIS

```
int pthread_condattr_destroy
    (
    pthread_condattr_t *pAttr  /* condition variable attributes */
    )
```

DESCRIPTION

This routine destroys the condition attribute object *pAttr*. It must not be reused until it is reinitialized.

RETURNS

Always returns zero.

ERRNO

None.

SEE ALSO

pthreadLib, **pthread_cond_init( )**, **pthread_condattr_init( )**

pthread_condattr_init( )

NAME

pthread_condattr_init( ) – initialize a condition attribute object (POSIX)

SYNOPSIS

```
int pthread_condattr_init
    (
    pthread_condattr_t *pAttr  /* condition variable attributes */
    )
```

DESCRIPTION

This routine initializes the condition attribute object *pAttr* and fills it with default values for the attributes.

RETURNS

On success zero; on failure a non-zero error code:

EINVAL

ERRNO

SEE ALSO

pthreadLib, **pthread_cond_init( )**, **pthread_condattr_destroy( )**

pthread_create( )

NAME **pthread_create( )** – create a thread (POSIX)

SYNOPSIS

```
int pthread_create
    (
    pthread_t *                pThread,  /* Thread ID (out) */
    const pthread_attr_t *     pAttr,    /* Thread attributes object */
    void * (*startRoutine)(void *),        /* Entry function */
    void *                     arg       /* Entry function argument */
    )
```

DESCRIPTION This routine creates a new thread and if successful writes its ID into the location pointed to by *pThread*. If *pAttr* is **NULL** then default attributes are used. The new thread executes *startRoutine* with *arg* as its argument.

The new thread's cancelability state and cancelability type are respectively set to **PTHREAD_CANCEL_ENABLE** and **PTHREAD_CANCEL_DEFERRED**.

RETURNS On success zero; on failure one of the following non-zero error codes:

EINVAL
can be returned when the value specified by *pAttr* is invalid, when a user-supplied stack address is provided but the stack size is invalid, and when the *pThread* parameter is null.

EAGAIN
can be returned when not enough memory is available to either create the thread or create a resource required for the thread.

EPERM
the explicit scheduling policy does not match the VxWorks scheduling policy currently in effect.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_exit( )**, **pthread_join( )**, **pthread_detach( )**

pthread_detach( )

NAME **pthread_detach( )** – dynamically detach a thread (POSIX)

SYNOPSIS

```
int pthread_detach
```

```
    (
    pthread_t thread  /* thread to detach */
    )
```

DESCRIPTION This routine puts the thread *thread* into the detached state. This prevents other threads from synchronizing on the termination of the thread using **pthread_join( )**.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ESRCH

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_join( )**

pthread_equal( )

NAME **pthread_equal( )** – compare thread IDs (POSIX)

SYNOPSIS

```
int pthread_equal
    (
    pthread_t t1,  /* thread one */
    pthread_t t2   /* thread two */
    )
```

DESCRIPTION Tests the equality of the two threads *t1* and *t2*.

RETURNS Non-zero if *t1* and *t2* refer to the same thread, otherwise zero.

ERRNO Not Available

SEE ALSO **pthreadLib**

pthread_exit( )

NAME **pthread_exit( )** – terminate a thread (POSIX)

SYNOPSIS

```
void pthread_exit
```

```
    (
    void *status  /* exit status */
    )
```

DESCRIPTION

This function terminates the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push( )** are executed in reverse order (the most recently added handler is executed first). Termination functions for thread-specific data are then called for all keys that have non-**NULL** values associated with them in the calling thread (see **pthread_key_create( )** for more details). Finally, execution of the calling thread is stopped.

The *status* argument is the return value of the thread and can be consulted from another thread using **pthread_join( )** unless this thread was detached (i.e. a call to **pthread_detach( )** had been made for it, or it was created in the detached state).

All threads that remain *joinable* at the time they exit should ensure that **pthread_join( )** is called on their behalf by another thread to reclaim the resources that they hold.

RETURNS

Does not return.

ERRNO

N/A

SEE ALSO

pthreadLib, **pthread_cleanup_push( )**, **pthread_detach( )**, **pthread_join( )**, **pthread_key_create( )**

pthread_getschedparam( )

NAME

pthread_getschedparam( ) – get value of schedparam attribute from a thread (POSIX)

SYNOPSIS

```
int pthread_getschedparam
    (
    pthread_t          thread,    /* thread                      */
    int                *pPolicy,  /* current policy (out)        */
    struct sched_param *pParam    /* current parameters (out)    */
    )
```

DESCRIPTION

This routine reads the current scheduling parameters and policy of the thread specified by *thread*. The information is returned via *pPolicy* and *pParam*.

Note that this routine actually always maps the current VxWorks scheduling policy on one of the two following POSIX scheduling policies: **SCHED_FIFO** or **SCHED_RR**. The **SCHED_OTHER** policy can therefore never be returned even if it has been set via **pthread_setschedparam( )**.

RETURNS

On success zero; on failure the **ESRCH** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getschedparam( ) pthread_attr_getschedpolicy( )**, **pthread_attr_setschedparam( ) pthread_attr_setschedpolicy( )**, **pthread_setschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_getspecific( )

NAME **pthread_getspecific( )** – get thread specific data (POSIX)

SYNOPSIS

```
void *pthread_getspecific
    (
    pthread_key_t key  /* thread specific data key */
    )
```

DESCRIPTION This routine returns the value associated with the thread specific data key *key* for the calling thread.

RETURNS The value associated with *key*, or **NULL**.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_key_create( )**, **pthread_key_delete( )**, **pthread_setspecific( )**

pthread_join( )

NAME **pthread_join( )** – wait for a thread to terminate (POSIX)

SYNOPSIS

```
int pthread_join
    (
    pthread_t thread,     /* thread to wait for           */
    void      **ppStatus  /* exit status of thread (out)  */
    )
```

DESCRIPTION This routine will block the calling thread until the thread specified by *thread* terminates, or is canceled. The thread must be in the joinable state, i.e. it cannot have been detached by a call to **pthread_detach( )**, or created in the detached state.

If *ppStatus* is not **NULL** and **pthread_join( )** returns successfully, when *thread* terminates its exit status will be stored in the specified location. The exit status will be either the value passed to **pthread_exit( )**, or **PTHREAD_CANCELED** if the thread was canceled or the thread was deleted by a VxWorks task.

Only one thread can wait for the termination of a given thread. If another thread is already waiting when this function is called an error will be returned (**EINVAL**).

If the calling thread passes its own ID in *thread*, the call will fail with the error **EDEADLK**.

NOTE All threads that remain *joinable* at the time they exit should ensure that **pthread_join( )** is called on their behalf by another thread to reclaim the resources that they hold.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ESRCH

EDEADLK

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_detach( )**, **pthread_exit( )**

pthread_key_create( )

NAME **pthread_key_create( )** – create a thread specific data key (POSIX)

SYNOPSIS

```
int pthread_key_create
    (
    pthread_key_t           *pKey,  /* thread specific data key     */
    void (*destructor)(void *)      /* destructor function          */
    )
```

DESCRIPTION This routine allocates a new thread specific data key. The key is stored in the location pointed to by *key*. The value initially associated with the returned key is **NULL** in all currently executing threads. If the maximum number of keys are already allocated, the function returns an error (**EAGAIN**).

The *destructor* parameter specifies a destructor function associated with the key. When a thread terminates via **pthread_exit( )**, or by cancellation, *destructor* is called with the value associated with the key in that thread as an argument. The destructor function is **not** called if that value is **NULL**. The order in which destructor functions are called at thread termination time is unspecified.

It is the user's responsibility to call **pthread_key_delete( )** when the memory associated with the key is no longer required, and to ensure that no threads access the key after it has been deleted. Failure to do this can return unexpected results, and can cause memory leaks.

RETURNS On success zero; on failure the **EAGAIN** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_getspecific( )**, **pthread_key_delete( )**, **pthread_setspecific( )**

pthread_key_delete( )

NAME **pthread_key_delete( )** – delete a thread specific data key (POSIX)

SYNOPSIS

```
int pthread_key_delete
    (
    pthread_key_t key  /* thread specific data key to delete */
    )
```

DESCRIPTION This routine deletes the thread specific data associated with *key*, and deallocates the key itself. It does not call any destructor associated with the key.

Any attempt to use key following the call to **pthread_key_delete( )** results in undefined behavior.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_key_create( )**

pthread_kill( )

NAME **pthread_kill( )** – send a signal to a thread (POSIX)

SYNOPSIS

```
int pthread_kill
    (
    pthread_t thread,  /* thread to signal */
    int       sig      /* signal to send */
    )
```

DESCRIPTION This routine sends signal number *sig* to the thread specified by *thread*. The signal is delivered and handled as described for the **kill( )** function.

RETURNS On success zero; on failure one of the following non-zero error codes: **ESRCH**, **EINVAL**

ERRNO
N/A

SEE ALSO
pthreadLib, **kill( )**, **pthread_sigmask( )**, **sigprocmask( )**, **sigaction( )**, **sigsuspend( )**, **sigwait( )**

pthread_mutex_destroy( )

NAME
pthread_mutex_destroy( ) – destroy a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_destroy
    (
    pthread_mutex_t *pMutex  /* POSIX mutex          */
    )
```

DESCRIPTION
This routine destroys a mutex object, freeing the resources it might hold. The mutex can be safely destroyed when unlocked. On VxWorks a thread may destroy a mutex that it owns (i.e. that the thread has locked). If the mutex is locked by an other thread this routine will return an error (**EBUSY**).

RETURNS
On success zero; on failure a non-zero error code:

EINVAL

EBUSY

ERRNO
N/A

SEE ALSO
pthreadLib, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semDelete( )**

pthread_mutex_getprioceiling( )

NAME
pthread_mutex_getprioceiling( ) – get the value of the prioceiling attribute of a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_getprioceiling
    (
    pthread_mutex_t *pMutex,       /* POSIX mutex                    */
    int             *pPrioceiling  /* current priority ceiling (out) */
    )
```

DESCRIPTION This function gets the current value of the prioceiling attribute of a mutex. Unless the mutex was created with a protocol attribute value of **PTHREAD_PRIO_PROTECT**, this value is meaningless.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_mutex_setprioceiling( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_setprioceiling( )**

pthread_mutex_init( )

NAME **pthread_mutex_init( )** – initialize mutex from attributes object (POSIX)

SYNOPSIS

```
int pthread_mutex_init
    (
    pthread_mutex_t           *pMutex,  /* POSIX mutex                  */
    const pthread_mutexattr_t *pAttr    /* mutex attributes             */
    )
```

DESCRIPTION This routine initializes the mutex object pointed to by *pMutex* according to the mutex attributes specified in *pAttr*. If *pAttr* is **NULL**, default attributes are used as defined in the POSIX specification. If *pAttr* is non-**NULL** then it is assumed to point to a mutex attributes object initialized by **pthread_mutexattr_init( )**, and those are the attributes used to create the mutex.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ERRNO N/A

SEE ALSO **pthreadLib**, **semLib**, **semMLib**, **pthread_mutex_destroy( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semMCreate( )**

pthread_mutex_lock( )

NAME **pthread_mutex_lock( )** – lock a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_lock
    (
    pthread_mutex_t *pMutex  /* POSIX mutex          */
    )
```

DESCRIPTION This routine locks the mutex specified by *pMutex*. If the mutex is currently unlocked, it becomes locked, and is said to be owned by the calling thread. In this case **pthread_mutex_lock( )** returns immediately.

If the mutex is already locked by another thread, **pthread_mutex_lock( )** blocks the calling thread until the mutex is unlocked by its current owner.

If it is already locked by the calling thread, pthread_mutex_lock will deadlock on itself and the thread will block indefinitely.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semTake( )**

pthread_mutex_setprioceiling( )

NAME **pthread_mutex_setprioceiling( )** – dynamically set the prioceiling attribute of a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_setprioceiling
    (
    pthread_mutex_t *pMutex,          /* POSIX mutex                   */
    int             prioceiling,      /* new priority ceiling          */
    int             *pOldPrioceiling  /* old priority ceiling (out)    */
    )
```

DESCRIPTION This function dynamically sets the value of the prioceiling attribute of a mutex. Unless the mutex was created with a protocol value of **PTHREAD_PRIO_PROTECT**, this function does nothing.

RETURNS
On success zero; on failure a non-zero error code:

EINVAL

EPERM

S_objLib_OBJ_ID_ERROR

S_semLib_NOT_ISR_CALLABLE

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_mutex_getprioceiling( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_setprioceiling( )**

pthread_mutex_trylock( )

NAME
pthread_mutex_trylock( ) – lock mutex if it is available (POSIX)

SYNOPSIS
```
int pthread_mutex_trylock
    (
    pthread_mutex_t *pMutex  /* POSIX mutex          */
    )
```

DESCRIPTION
This routine locks the mutex specified by *pMutex*. If the mutex is currently unlocked, it becomes locked and owned by the calling thread. In this case **pthread_mutex_trylock( )** returns immediately.

If the mutex is already locked by another thread, **pthread_mutex_trylock( )** returns immediately with the error code **EBUSY**.

RETURNS
On success zero; on failure a non-zero error code:

EINVAL

EBUSY

ERRNO
N/A

SEE ALSO
pthreadLib, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semTake( )**

pthread_mutex_unlock( )

NAME

pthread_mutex_unlock( ) – unlock a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_unlock
    (
    pthread_mutex_t *pMutex
    )
```

DESCRIPTION

This routine unlocks the mutex specified by *pMutex*. If the calling thread is not the current owner of the mutex, **pthread_mutex_unlock( )** returns with the error code **EPERM**.

RETURNS

On success zero; on failure a non-zero error code:

EINVAL

EPERM

S_objLib_OBJ_ID_ERROR

S_semLib_NOT_ISR_CALLABLE

ERRNO

N/A

SEE ALSO

pthreadLib, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semGive( )**

pthread_mutexattr_destroy( )

NAME

pthread_mutexattr_destroy( ) – destroy mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_destroy
    (
    pthread_mutexattr_t *pAttr  /* mutex attributes */
    )
```

DESCRIPTION

This routine destroys a mutex attribute object. The mutex attribute object must not be reused until it is reinitialized.

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO

N/A

SEE ALSO **pthreadLib**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_getprioceiling( )

NAME **pthread_mutexattr_getprioceiling( )** – get the current value of the prioceiling attribute in a mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_getprioceiling
    (
    pthread_mutexattr_t *pAttr,        /* mutex attributes                */
    int                 *pPrioceiling  /* current priority ceiling (out) */
    )
```

DESCRIPTION This function gets the current value of the prioceiling attribute in a mutex attributes object. Unless the value of the protocol attribute is **PTHREAD_PRIO_PROTECT**, this value is ignored.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_getprotocol( )

NAME **pthread_mutexattr_getprotocol( )** – get value of protocol in mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_getprotocol
    (
    pthread_mutexattr_t *pAttr,     /* mutex attributes             */
    int                 *pProtocol  /* current protocol (out)       */
    )
```

DESCRIPTION This function gets the current value of the protocol attribute in a mutex attributes object.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_init( )

NAME **pthread_mutexattr_init( )** – initialize mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_init
    (
    pthread_mutexattr_t *pAttr  /* mutex attributes */
    )
```

DESCRIPTION This routine initializes the mutex attribute object *pAttr* and fills it with default values for the attributes:

Mutex Protocol
PTHREAD_PRIO_INHERIT - the priority of the owner thread is temporarily raised if a higher priority thread is blocked on the mutex.

Mutex Priority Ceiling
0 - lowest priority.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_setprioceiling( )

NAME **pthread_mutexattr_setprioceiling( )** – set prioceiling attribute in mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_setprioceiling
    (
    pthread_mutexattr_t *pAttr,      /* mutex attributes     */
    int                  prioceiling  /* new priority ceiling */
    )
```

DESCRIPTION
This function sets the value of the prioceiling attribute in a mutex attributes object. Unless the protocol attribute is set to **PTHREAD_PRIO_PROTECT**, this attribute is ignored.

RETURNS
On success zero; on failure the **EINVAL** error code.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_setprotocol( )

NAME
pthread_mutexattr_setprotocol( ) – set protocol attribute in mutex attribute object (POSIX)

SYNOPSIS
```
int pthread_mutexattr_setprotocol
    (
    pthread_mutexattr_t *pAttr,   /* mutex attributes     */
    int                  protocol  /* new protocol         */
    )
```

DESCRIPTION
This function selects the locking protocol to be used when a mutex is created using this attributes object. The protocol to be selected is either **PTHREAD_PRIO_INHERIT** or **PTHREAD_PRIO_PROTECT**.

RETURNS
On success zero; on failure a non-zero error code:

EINVAL

ENOTSUP

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutex_init( )**

pthread_once( )

NAME

pthread_once( ) – dynamic package initialization (POSIX)

SYNOPSIS

```
int pthread_once
    (
    pthread_once_t * pOnceControl,      /* once control location         */
    void             (*initFunc)(void)  /* function to call              */
    )
```

DESCRIPTION

This routine provides a mechanism to ensure that one, and only one call to a user specified initialization function will occur. This allows all threads in a system to attempt initialization of some feature they need to use, without any need for the application to explicitly prevent multiple calls.

When a thread makes a call to **pthread_once( )**, the first thread to call it with the specified control variable, *pOnceControl*, will result in a call to *initFunc*, but subsequent calls will not. The *pOnceControl* parameter determines whether the associated initialization routine has been called. The *initFunc* function is complete when **pthread_once( )** returns.

The function **pthread_once( )** is not a cancellation point; however, if the function *initFunc* is a cancellation point, and the thread is canceled while executing it, the effect on *pOnceControl* is the same as if **pthread_once( )** had never been called.

CAVEAT

If the initialization function does not return then all threads calling **pthread_once( )** with the same control variable will stay blocked as well. It is therefore imperative that the initialization function always returns.

WARNING

If *pOnceControl* has automatic storage duration or is not initialized to the value **PTHREAD_ONCE_INIT**, the behavior of **pthread_once( )** is undefined.

The constant **PTHREAD_ONCE_INIT** is defined in the **pthread.h** header file.

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO

None

SEE ALSO

pthreadLib

pthread_self( )

NAME **pthread_self( )** – get the calling thread's ID (POSIX)

SYNOPSIS

```
pthread_t pthread_self (void)
```

DESCRIPTION This function returns the calling thread's ID.

If the caller is a native VxWorks task it will be given a POSIX thread persona.

RETURNS Calling thread's ID.

ERRNO Not Available

SEE ALSO **pthreadLib**

pthread_setcancelstate( )

NAME **pthread_setcancelstate( )** – set cancellation state for calling thread (POSIX)

SYNOPSIS

```
int pthread_setcancelstate
    (
    int state,     /* new state             */
    int *oldstate  /* old state (out)       */
    )
```

DESCRIPTION This routine sets the cancellation state for the calling thread to *state*, and, if *oldstate* is not **NULL**, returns the old state in the location pointed to by *oldstate*.

The state can be one of the following:

PTHREAD_CANCEL_ENABLE
Enable thread cancellation.

PTHREAD_CANCEL_DISABLE
Disable thread cancellation (i.e. thread cancellation requests are ignored).

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_cancel( )**, **pthread_setcanceltype( )**, **pthread_testcancel( )**

pthread_setcanceltype( )

NAME

pthread_setcanceltype( ) – set cancellation type for calling thread (POSIX)

SYNOPSIS

```
int pthread_setcanceltype
    (
    int type,     /* new type             */
    int *oldtype  /* old type (out)       */
    )
```

DESCRIPTION

This routine sets the cancellation type for the calling thread to *type*. If *oldtype* is not **NULL**, then the old cancellation type is stored in the location pointed to by *oldtype*.

Possible values for *type* are:

PTHREAD_CANCEL_ASYNCHRONOUS
Any cancellation request received by this thread will be acted upon as soon as it is received.

PTHREAD_CANCEL_DEFERRED
Cancellation requests received by this thread will be deferred until the next cancellation point is reached.

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO

N/A

SEE ALSO

pthreadLib, **pthread_cancel( )**, **pthread_setcancelstate( )**, **pthread_testcancel( )**

pthread_setschedparam( )

NAME

pthread_setschedparam( ) – dynamically set schedparam attribute for a thread (POSIX)

SYNOPSIS

```
int pthread_setschedparam
    (
    pthread_t                 thread,  /* thread                */
    int                       policy,  /* new policy            */
    const struct sched_param *pParam   /* new parameters        */
    )
```

DESCRIPTION

This routine will set the scheduling parameters (*pParam*) and policy (*policy*) for the thread specified by *thread*.

In VxWorks the scheduling policy is global and not set on a per-thread basis; if the selected policy is one of **SCHED_FIFO** or **SCHED_RR** and this does not match the current VxWorks scheduling policy then this function will return an error (**EPERM**). If the *policy* parameter is set to **SCHED_OTHER**, which always matches the active scheduling policy, only the thread's priority will be changed.

RETURNS On success zero; on failure one of the following non-zero error codes: **EPERM**, **ESRCH** (invalid task ID), **EINVAL** (scheduling priority is outside valid range)

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**, **pthread_getschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_setspecific( )

NAME **pthread_setspecific( )** – set thread specific data (POSIX)

SYNOPSIS

```
int pthread_setspecific
    (
    pthread_key_t key,    /* thread specific data key     */
    const void    *value  /* new value                    */
    )
```

DESCRIPTION Sets the value of the thread specific data associated with *key* to *value* for the calling thread.

RETURNS On success zero; on failure a non-zero error code:

EINVAL

ENOMEM

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_getspecific( )**, **pthread_key_create( )**, **pthread_key_delete( )**

pthread_sigmask( )

NAME **pthread_sigmask( )** – change and/or examine calling thread's signal mask (POSIX)

SYNOPSIS

```
int pthread_sigmask
```

```
(
int               how,  /* method for changing set      */
const sigset_t * set,  /* new set of signals           */
sigset_t *        oset  /* old set of signals           */
)
```

DESCRIPTION

This routine changes the signal mask for the calling thread as described by the *how* and *set* arguments. If *oset* is not **NULL**, the previous signal mask is stored in the location pointed to by it.

The value of *how* indicates the manner in which the set is changed and consists of one of the following defined in **signal.h**:

SIG_BLOCK
The resulting set is the union of the current set and the signal set pointed to by *set*.

SIG_UNBLOCK
The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.

SIG_SETMASK
The resulting set is the signal set pointed to by *oset*.

RETURNS

On success zero; on failure a **EINVAL** error code is returned.

ERRNO

N/A

SEE ALSO

pthreadLib, **kill( )**, **pthread_kill( )**, **sigprocmask( )**, **sigaction( )**, **sigsuspend( )**, **sigwait( )**

pthread_testcancel( )

NAME

pthread_testcancel( ) – create a cancellation point in the calling thread (POSIX)

SYNOPSIS

```
void pthread_testcancel (void)
```

DESCRIPTION

This routine creates a cancellation point in the calling thread. It has no effect if cancellation is disabled (i.e. the cancellation state has been set to **PTHREAD_CANCEL_DISABLE** using the **pthread_setcancelstate( )** function).

If cancellation is enabled, the cancellation type is **PTHREAD_CANCEL_DEFERRED** and a cancellation request has been received, then this routine will call **pthread_exit( )** with the exit status set to **PTHREAD_CANCELED**. If any of these conditions is not met, then the routine does nothing.

RETURNS

N/A

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_cancel( )**, **pthread_setcancelstate( )**, **pthread_setcanceltype( )**

ptyDevCreate( )

NAME **ptyDevCreate( )** – create a pseudo terminal

SYNOPSIS

```
STATUS ptyDevCreate
    (
    char *name,      /* name of pseudo terminal */
    int  rdBufSize,  /* size of terminal read buffer */
    int  wrtBufSize  /* size of write buffer */
    )
```

DESCRIPTION This routine creates a master and slave device which can then be opened by the master and slave processes. The master process simulates the "hardware" side of the driver, while the slave process is the application program that normally talks to a *tty* driver. Data written to the master device can then be read on the slave device, and vice versa.

RETURNS **OK**, or **ERROR** if memory is insufficient.

ERRNO

S_ioLib_NO_DRIVER (**ENXIO**)
The **ptyDrv** driver is not installed.

S_iosLib_DUPICATE_DEVICE_NAME (**EINVAL**)
The device name is already in use.

SEE ALSO **ptyDrv**

ptyDevRemove( )

NAME **ptyDevRemove( )** – destroy a pseudo terminal

SYNOPSIS

```
STATUS ptyDevRemove
    (
    char * pName  /* name of pseudo terminal to remove */
    )
```

DESCRIPTION This routine removes an existing master and slave device and releases all allocated memory. It will close any open files using either device.

RETURNS **OK**, or **ERROR** if terminal not found

ERRNO **S_ioLib_NO_DRIVER** (**ENXIO**)
The **ptyDrv** is not installed.

SEE ALSO **ptyDrv**

ptyDrv( )

NAME **ptyDrv( )** – initialize the pseudo-terminal driver

SYNOPSIS

```
STATUS ptyDrv (void)
```

DESCRIPTION This routine initializes the pseudo-terminal driver. It must be called before any other routine in this module.

RETURNS **OK**, or **ERROR** if the master or slave devices cannot be installed.

ERRNO N/A

SEE ALSO **ptyDrv**

putenv( )

NAME **putenv( )** – set an environment variable

SYNOPSIS

```
STATUS putenv
    (
    char *pEnvString  /* string to add to env */
    )
```

DESCRIPTION This routine sets an environment variable to a value by altering an existing variable or creating a new one. The parameter points to a string of the form "variableName=value". Unlike the UNIX implementation, the string passed as a parameter is copied to a private buffer.

RETURNS **OK**, or **ERROR** if space cannot be malloc'd.

ERRNOS **S_memLib_NOT_ENOUGH_MEMORY**
There is no free block large enough to satisfy the allocation request.

SEE ALSO **envLib**, **envLibInit( )**, **getenv( )**

pwd( )

NAME **pwd( )** – print the current default directory

SYNOPSIS

```
void pwd (void)
```

DESCRIPTION This command displays the current working device/directory.

NOTE This is a target resident function, which manipulates the target I/O system. It must be preceded with the @ letter if executed from the Host Shell (windsh), which has a built-in command of the same name that operates on the Host's I/O system.

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrFsLib**, **cd( )**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

quiccEngineDrvCtrlShow( )

NAME **quiccEngineDrvCtrlShow( )** – place holder just prints out control structure ptr

SYNOPSIS

```
int quiccEngineDrvCtrlShow
    (
    VXB_DEVICE_ID pInst
    )
```

DESCRIPTION none

RETURNS N/A

ERRNO

SEE ALSO **quiccEngineUtils**

quiccEngineRegister( )

NAME **quiccEngineRegister( )** – register quiccEngine driver

SYNOPSIS

```
void quiccEngineRegister(void)
```

DESCRIPTION This routine registers the quiccEngine driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO **quiccEngineUtils**

r0( )

NAME **r0( )** – return the contents of register **r0** (also **r1** - **r14**) (ARM)

SYNOPSIS

```
int r0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of register **r0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for registers (**r1** - **r14**): **r1( )** - **r14( )**.

RETURNS The contents of register **r0** (or the requested register).

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

r0( )

NAME

r0( ) – return the contents of general register **r0** (also **r1**-`r15') (SH)

SYNOPSIS

```
int r0
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **r0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all general registers (**r1** - **r15**): **r1( )** - **r15( )**.

RETURNS

The contents of register r0 (or the requested register).

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

raise( )

NAME

raise( ) – send a signal to the caller's task

SYNOPSIS

```
int raise
    (
    int signo  /* signal to send to caller's task */
    )
```

DESCRIPTION

This routine sends the signal *signo* to the task invoking the call.

RETURNS

OK (0), or **ERROR** (-1) if the signal number or task ID is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib, **taskRaise( )**

ramDevCreate( )

NAME

ramDevCreate( ) – create a RAM disk device

SYNOPSIS

```
BLK_DEV* ramDevCreate
    (
    char *ramAddr,      /* where it is in memory (0 = malloc) */
    int  bytesPerBlk,   /* number of bytes per block */
    int  blksPerTrack,  /* number of blocks per track */
    int  nBlocks,       /* number of blocks on this device */
    int  blkOffset      /* no. of blks to skip at start of device */
    )
```

DESCRIPTION

This routine creates a RAM disk device.

Memory for the RAM disk can be pre-allocated separately; if so, the *ramAddr* parameter should be the address of the pre-allocated device memory. Or, memory can be automatically allocated with **malloc( )** by setting *ramAddr* to zero.

The *bytesPerBlk* parameter specifies the size of each logical block on the RAM disk. If *bytesPerBlk* is zero, 512 is used.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the RAM disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (i.e., the disk is defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, a default size is used. The default is calculated using a total disk size of either 51,200 bytes or one-half of the size of the largest memory area available, whichever is less. This default disk size is then divided by *bytesPerBlk* to determine the number of blocks.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the RAM disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.) This offset value is typically useful only if a specific address is given for *ramAddr*. Normally, *blkOffset* is 0.

FILE SYSTEMS

Once the device has been created, it must be associated with a name and a file system (dosFs, hrfs, or rawFs). This is accomplished in a two step process. The **ramDevCreate( )** 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 **ramDrv** driver. The **BLK_DEV** structure address should be passed to an XBD wrapper via **xbdBlkDevCreate( )** along with the name of the device. XBDs are the new and preferred method for interfacing with file systems.

After the XBD wrapper is created, the file system framework will attempt to identify the type of file system instantiated on the device. If it can not be identified, then it is instantiated with rawFs.

The desired file system (dosFs or hrfs) can be instantiated on the ram drive using either **dosFsVolFormat( )**, **dosfsDiskFormat( )**,**hrfsFormat( )**, or **hrfsDiskFormat( )**. The ram drive to be formatted is identified by the name of the device given in the XBD wrapper.

EXAMPLE

In the following example, a 208-Kbyte RAM disk is created with automatically allocated memory, 512-byte blocks, 32 blocks per track, and no block offset. The device is then initialized for use with dosFs and assigned the name "/**ramDrv**":

```
BLK_DEV *pBlkDev;

pBlkDev = ramDevCreate (NULL,  512,  32,  416,  0);
xbdBlkDevCreate (pBlkDev, "/ramDrv");
dosFsVolFormat ("/ramDrv:0", DOS_OPT_BLANK, NULL);
```

The names used in **xbdBlkDevCreate( )** and **dosFsVolFormat( )** are slightly different on purpose. The ":0" is appended to "/**ramDrv**" by **xbdBlkDevCreate( )** and represents the whole (unpartitioned) disk.

If the RAM disk memory already contains a disk image created elsewhere, the first argument to **ramDevCreate( )** should be the address in memory, and the formatting parameters -- *bytesPerBlk*, *blksPerTrack*, *nBlocks*, and *blkOffset* -- must be identical to those used when the image was created. For example:

```
pBlkDev = ramDevCreate (0xc0000, 512, 32, 416, 0);
```

In this case, the file system does not have to be explicitly created as the file system framework will probe the ram drive to determine the type of file system previously instantiated on it. The detected file system will be automatically re-instantiated on the device. This procedure is useful if a RAM disk is to be created at the same address used in a previous boot of VxWorks. The contents of the RAM disk will then be preserved.

If no known file system was detected, the ram drive will default to rawFs.

RETURNS

A pointer to a block device structure (**BLK_DEV**) or **NULL** if memory cannot be allocated for the device structure or for the RAM disk.

ERRNO

N/A.

SEE ALSO

ramDrv, **xbdBlkDevCreate( )**, **dosFsVolFormat( )**, **hrfsFormat( )**

ramDiskDevCreate( )

NAME

ramDiskDevCreate( ) – Initialize a RAM Disk device

SYNOPSIS

```
CBIO_DEV_ID ramDiskDevCreate
```

```
    (
    char *pRamAddr,     /* where it is in memory (0 = malloc)     */
    int  bytesPerBlk,   /* number of bytes per block              */
    int  blksPerTrack,  /* number of blocks per track             */
    int  nBlocks,       /* number of blocks on this device        */
    int  blkOffset      /* no. of blks to skip at start of device */
    )
```

DESCRIPTION This function creates a compact RAM-Disk device that can be directly utilized by **dosFsLib**, without the intermediate disk cache. It can be used for non-volatile RAM as well as volatile RAM disks.

The RAM size is specified in terms of total number of blocks in the device and the block size in bytes. The minimal block size is 32 bytes. If *pRamAddr* is **NULL**, space will be allocated from the default memory pool.

CAVEAT When used with NV-RAM, this module can not eliminate mid-block write interruption, which may cause file system corruption not existent in common disk drives.

RETURNS a CBIO handle that can be directly used by **dosFsDevCreate( )** or **NULL** if the requested amount of RAM is not available.

ERRNO Not Available

SEE ALSO **ramDiskCbio**, **dosFsDevCreate( )**.

ramDrv( )

NAME **ramDrv( )** – prepare a RAM disk driver for use (optional)

SYNOPSIS

```
STATUS ramDrv (void)
```

DESCRIPTION This routine performs no real function, except to provide compatibility with earlier versions of **ramDrv** and to parallel the initialization function found in true disk device drivers. It also is used in **usrConfig.c** to link in the RAM disk driver when building VxWorks. It is automatically called when VxWorks is configured with the **INCLUDE_RAMDRV** component.

RETURNS **OK**, always.

ERRNO N/A.

SEE ALSO **ramDrv**

rawFsDevInit( )

NAME **rawFsDevInit( )** – associate a block device with raw volume functions

SYNOPSIS

```
RAW_VOL_DESC *rawFsDevInit
    (
    char      * pVolName,  /* volume name to be used with iosDevAdd */
    device_t    xbd        /* XBD device */
    )
```

DESCRIPTION This routine takes a block device created by a device driver and defines it as a raw file system volume. As a result, when high-level I/O operations, such as **open( )** and **write( )** are performed, on the device, the calls will be routed through **rawFsLib**.

This routine associates *pVolName* with a device and installs it in the VxWorks I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the raw library during **rawFsInit( )**. (The driver number is kept in the global variable **rawFsDrvNum**.)

The *xbd* is a device_t referring to an XBD device which represents the backing media for this rawFs. The XBD device will not be accessed until an I/O operation occurs on the file system.

RETURNS A pointer to the volume descriptor (**RAW_VOL_DESC**), or **NULL** if there is an error.

ERRNO Not Available

SEE ALSO **rawFsLib**

rawFsInit( )

NAME **rawFsInit( )** – prepare to use the raw volume library

SYNOPSIS

```
STATUS rawFsInit
    (
    int maxFiles  /* max no. of simultaneously open files */
    )
```

DESCRIPTION This routine initializes the raw volume library. It must be called exactly once, before any other routine in the library. The argument specifies the number of file descriptors that may be open at once. This routine allocates and sets up the necessary memory structures and initializes semaphores.

This routine also installs raw volume library routines in the VxWorks I/O system driver table. The driver number assigned to **rawFsLib** is placed in the global variable

rawFsDrvNum. This number will later be associated with system file descriptors opened to rawFs devices.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **rawFsLib**

rawPerfDemo( )

NAME **rawPerfDemo( )** – entry point for the VxWorks/SMP raw performance demo

SYNOPSIS

```
STATUS rawPerfDemo (void)
```

DESCRIPTION This routine is the entry point for the VxWorks/SMP raw performance demo. It is typically called from the target shell.

This function will create N worker tasks; N = number of CPUs currently enabled in the system. Aggregate raw performance figures are obtained as described in the module description. The aggregate raw performance data is plotted in real-time on an ASCII character graph.

See the module description for more information.

RETURNS **ERROR** if failed to create worker tasks, otherwise **OK** is returned.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
Out of memory for creation of worker tasks

SEE ALSO **rawPerfDemo**

read( )

NAME **read( )** – read bytes from a file or device

SYNOPSIS

```
int read
    (
    int    fd,       /* file descriptor from which to read   */
    char * buffer,   /* pointer to buffer to receive bytes   */
    size_t maxbytes  /* max no. of bytes to read into buffer */
    )
```

DESCRIPTION

This routine reads a number of bytes (less than or equal to *maxbytes*) from a specified file descriptor and places them in *buffer*. It calls the device driver to do the work.

RETURNS

The number of bytes read (between 1 and *maxbytes*, 0 if end of file), or **ERROR** if the file descriptor does not exist, the driver does not have a read routines, or the driver returns **ERROR**. If the driver does not have a read routine, errno is set to **ENOTSUP**.

ERRNO

EBADF
Bad file descriptor number.

ENOTSUP
Device driver does not support the read command.

ENXIO
Device and its driver are removed. **close( )** should be called to release this file descriptor.

Other
Other errors reported by device driver.

SEE ALSO

ioLib

readdir( )

NAME

readdir( ) – read one entry from a directory (POSIX)

SYNOPSIS

```
struct dirent *readdir
    (
    DIR *pDir  /* pointer to directory descriptor */
    )
```

DESCRIPTION

This routine obtains directory entry data for the next file from an open directory. The *pDir* parameter is the pointer to a directory descriptor (DIR) which was returned by a previous **opendir( )**.

This routine returns a pointer to a **dirent** structure which contains the name of the next file. Empty directory entries and MS-DOS volume label entries are not reported. The name of the file (or subdirectory) described by the directory entry is returned in the **d_name** field of the **dirent** structure. The name is a single null-terminated string.

The returned **dirent** pointer will be **NULL**, if it is at the end of the directory or if an error occurred. Because there are two conditions which might cause **NULL** to be returned, the task's error number (**errno**) must be used to determine if there was an actual error. Before calling **readdir( )**, set **errno** to **OK**. If a **NULL** pointer is returned, check the new value of **errno**. If **errno** is still **OK**, the end of the directory was reached; if not, **errno** contains the error code for an actual error which occurred.

RETURNS A pointer to a **dirent** structure, or **NULL** if there is an end-of-directory marker or error from the IO system.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

Other
Other errors reported by device driver.

SEE ALSO **dirLib**, **opendir( )**, **readdir_r( )**, **closedir( )**, **rewinddir( )**, **ls( )**

readdir_r( )

NAME **readdir_r( )** – read one entry from a directory (POSIX)

SYNOPSIS

```
int readdir_r
    (
    DIR           *pDir,     /* pointer to directory descriptor */
    struct dirent *entry,    /* pointer to directory entry */
    struct dirent **result   /* pointer to pointer to result of read */
    )
```

DESCRIPTION This routine obtains directory entry data for the next file from an open directory. The *pDir* parameter is the pointer to a directory descriptor (DIR) which was returned by a previous **opendir( )**.

The caller must allocate storage pointed to by *entry* to be large enough for a dirent structure with an array of char d_name member containing at least **NAME_MAX**.

On successful return, the pointer returned at **result* will be the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value **NULL**.

RETURNS zero if successful or an error number to indicate failure.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

Other
Other errors reported by device driver.

SEE ALSO **dirLib**, **opendir( )**, **readdir( )**, **closedir( )**, **rewinddir( )**, **ls( )**

realloc( )

NAME **realloc( )** – reallocate a block of memory (ANSI)

SYNOPSIS

```
void * realloc
    (
    void * pBlock,  /* block to reallocate */
    size_t newSize  /* new block size */
    )
```

DESCRIPTION This routine changes the size of a specified block of memory and returns a pointer to the new block of memory. The contents that fit inside the new size (or old size if smaller) remain unchanged. The memory alignment of the new block is not guaranteed to be the same as the original block.

When the **INCLUDE_MEM_MGR_FULL** component is included this funcition changes to an optimized implementation that attempts to resize the existing block.

If *pBlock* is **NULL**, this call is equivalent to **malloc( )**.

If *newSize* is set to zero and *pBlock* points to a valid allocated block, this call is equivalent to **free( )**.

RETURNS A pointer to the new block of memory, **NULL** if the call fails or if *newSize* is equal to zero.

ERRNO Possible errnos generated by this routine include:

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO **memPartLib**, **memPartRealloc( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: General Utilities (***stdlib.h***)*

reboot( )

NAME

reboot( ) – reset network devices and transfer control to boot ROMs

SYNOPSIS

```
void reboot
    (
    int startType  /* how the boot ROMS will reboot */
    )
```

DESCRIPTION

This routine returns control to the boot ROMs after calling a series of preliminary shutdown routines that have been added via **rebootHookAdd( )**, including routines to reset all network devices. After calling the shutdown routines, interrupts are locked, all caches are cleared, and control is transferred to the boot ROMs.

The bit values for *startType* are defined in **sysLib.h**:

BOOT_NORMAL (0x00)
causes the system to go through the countdown sequence and try to reboot VxWorks automatically. Memory is not cleared.

BOOT_NO_AUTOBOOT (0x01)
causes the system to display the VxWorks boot prompt and wait for user input to the boot ROM monitor. Memory is not cleared.

BOOT_CLEAR (0x02)
the same as **BOOT_NORMAL**, except that memory is cleared.

BOOT_QUICK_AUTOBOOT (0x04)
the same as **BOOT_NORMAL**, except the countdown is shorter.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

rebootLib, **sysToMonitor( )**, **rebootHookAdd( )**, **windsh**, the VxWorks programmer guides, and the IDE and host tools guides.

rebootHookAdd( )

NAME

rebootHookAdd( ) – add a routine to be called at reboot

SYNOPSIS

```
STATUS rebootHookAdd
    (
    FUNCPTR rebootHook  /* routine to be called at reboot */
    )
```

DESCRIPTION

This routine adds the specified routine to a list of routines to be called when VxWorks is rebooted. The specified routine should be declared as follows:

```
void rebootHook
    (
    int startType   /* startType is passed to all hooks */
    )
```

Reboot hooks will be called in the order they were added, when the **reboot( )** function is executed.

Reboot hooks must follow similar restrictions as with interrupt service routines (ISRs). Reboot hooks must not invoke kernel service routines that may block. For example, calling **free( )** or **semTake( )** while the semaphore is not available, would cause the caller to block. Blocking calls within the reboot hooks may causes the reboot process to **reschedule( )** or hang, potentially leaving the system in an undefined state.

RETURNS

OK, or **ERROR** if memory is insufficient.

ERRNO

Not Available

SEE ALSO

rebootLib, **reboot( )**

reld( )

NAME

reld( ) – reload an object module (shell command)

SYNOPSIS

```
MODULE_ID reld
    (
    void * nameOrId,  /* name or ID of the object module file */
    int    options    /* options used for unloading           */
    )
```

DESCRIPTION

This routine unloads a specified object module from the system, and then calls **loadModule( )** to load a new copy of the same name.

If the file was originally loaded using a complete pathname, then **reld( )** will use the complete name to locate the file. If the file was originally loaded using a partial pathname, then the current working directory must be changed to the working directory in use at the time of the original load.

Valid values for the options parameter are the same as those allowed for the function **unld( )**.

This routine is a **shell command**. That is, it is designed to be used only in the shell, and not in code running on the target. In future releases, calling **reld( )** directly from code may not be supported.

RETURNS
A module ID (type **MODULE_ID**), or **NULL**.

ERRNO
Not Available

SEE ALSO
usrLib, **loadLib**, **unld( )**, **ld( )**, the VxWorks programmer guides.

rename( )

NAME
rename( ) – change the name of a file

SYNOPSIS
```
int rename
    (
    const char *oldname,  /* name of file to rename          */
    const char *newname   /* name with which to rename file */
    )
```

DESCRIPTION
This routine changes the name of a file from *oldfile* to *newfile*.

NOTE
Only certain devices support **rename( )**. To confirm that your device supports it, consult the respective **xxDrv** or xxFs listings to verify that ioctl FIORENAME exists. For example, dosFs, HRFS and NFS support **rename( )**, but **netDrv** does not.

RETURNS
OK, or **ERROR** if the file could not be opened or renamed.

ERRNO

ENOENT
Either oldname or newname is an empty string.

ELOOP
Circular symbolic link, too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in path.

ENOSYS
Device driver does not support the symlink ioctl command.

others
Other errors reported by device driver.

SEE ALSO
fsPxLib

repeat( )

NAME
: **repeat( )** – spawn a task to call a function repeatedly

SYNOPSIS

```
int repeat
    (
    FAST int     n,     /* no. of times to call func (0=forever) */
    FAST FUNCPTR func,  /* function to call repeatedly         */
    int          arg1,  /* first of eight args to pass to func */
    int          arg2,
    int          arg3,
    int          arg4,
    int          arg5,
    int          arg6,
    int          arg7,
    int          arg8
    )
```

DESCRIPTION
: This command spawns a task that calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly, or until the spawned task is deleted.

NOTE
: The task is spawned using **sp( )**. See the description of **sp( )** for details about priority, options, stack size, and task ID.

RETURNS
: A task ID, or **ERROR** if the task cannot be spawned.

ERRNO
: **sp( )** errnos.

SEE ALSO
: **usrLib**, **repeatRun( )**, **sp( )**, the VxWorks programmer guides.

repeatRun( )

NAME
: **repeatRun( )** – call a function repeatedly

SYNOPSIS

```
void repeatRun
    (
    FAST int     n,     /* no. of times to call func (0=forever) */
    FAST FUNCPTR func,  /* function to call repeatedly         */
    int          arg1,  /* first of eight args to pass to func */
    int          arg2,
    int          arg3,
    int          arg4,
    int          arg5,
    int          arg6,
    int          arg7,
    int           arg8
    )
```

DESCRIPTION This command calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly.

Normally, this routine is called only by **repeat( )**, which spawns it as a task.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **repeat( )**, the VxWorks programmer guides.

rewinddir( )

NAME **rewinddir( )** – reset position to the start of a directory (POSIX)

SYNOPSIS

```
void rewinddir
    (
    DIR *pDir  /* pointer to directory descriptor */
    )
```

DESCRIPTION This routine resets the position pointer in a directory descriptor (DIR). The *pDir* parameter is the directory descriptor pointer that was returned by **opendir( )**.

As a result, the next **readdir( )** will cause the current directory data to be read in again, as if an **opendir( )** had just been performed. Any changes in the directory that have occurred since the initial **opendir( )** will now be visible. The first entry in the directory will be returned by the next **readdir( )**.

RETURNS N/A

ERRNO N/A.

SEE ALSO **dirLib**, **opendir( )**, **readdir( )**, **closedir( )**

rindex( )

NAME

rindex( ) – find the last occurrence of a character in a string

SYNOPSIS

```
char *rindex
    (
    FAST const char * s,  /* string in which to find character */
    int               c   /* character to find in string       */
    )
```

DESCRIPTION

This routine finds the last occurrence of character *c* in string *s*.

RETURNS

A pointer to *c*, or **NULL** if *c* is not found.

ERRNO

N/A

SEE ALSO

bLib

rm( )

NAME

rm( ) – remove a file

SYNOPSIS

```
STATUS rm
    (
    const char * fileName  /* name of file to remove */
    )
```

DESCRIPTION

This command is provided for UNIX similarity. It simply calls **remove( )**.

RETURNS

OK, or **ERROR** if the file cannot be removed.

ERRNO

Not Available

SEE ALSO

usrFsLib, **remove( )**, the VxWorks programmer guides.

rmdir( )

NAME

rmdir( ) – remove a directory

SYNOPSIS

```
STATUS rmdir
    (
    const char * dirName  /* name of directory to remove */
    )
```

DESCRIPTION

This command removes an existing directory from a hierarchical file system. The *dirName* string specifies the name of the directory to be removed, and may be either a full or relative pathname.

This call is supported by the VxWorks NFS and dosFs file systems.

RETURNS

OK, or **ERROR** if the directory cannot be removed.

ERRNO

Not Available

SEE ALSO

usrFsLib, **mkdir( )**, the VxWorks programmer guides.

rngBufGet( )

NAME

rngBufGet( ) – get characters from a ring buffer

SYNOPSIS

```
int rngBufGet
    (
    FAST RING_ID rngId,    /* ring buffer to get data from      */
    char         *buffer,  /* pointer to buffer to receive data */
    int          maxbytes  /* maximum number of bytes to get    */
    )
```

DESCRIPTION

This routine copies bytes from the ring buffer *rngId* into *buffer*. It copies as many bytes as are available in the ring, up to *maxbytes*. The bytes copied will be removed from the ring.

RETURNS

The number of bytes actually received from the ring buffer; it may be zero if the ring buffer is empty at the time of the call.

ERRNO

N/A.

SEE ALSO

rngLib

rngBufPut( )

NAME

rngBufPut( ) – put bytes into a ring buffer

SYNOPSIS

```
int rngBufPut
    (
    FAST RING_ID rngId,    /* ring buffer to put data into  */
    char         *buffer,  /* buffer to get data from       */
    int          nbytes    /* number of bytes to try to put */
    )
```

DESCRIPTION

This routine puts bytes from *buffer* into ring buffer *ringId*. The specified number of bytes will be put into the ring, up to the number of bytes available in the ring.

RETURNS

The number of bytes actually put into the ring buffer; it may be less than number requested, even zero, if there is insufficient room in the ring buffer at the time of the call.

ERRNO

N/A.

SEE ALSO

rngLib

rngCreate( )

NAME

rngCreate( ) – create an empty ring buffer

SYNOPSIS

```
RING_ID rngCreate
    (
    int nbytes  /* number of bytes in ring buffer */
    )
```

DESCRIPTION

This routine creates a ring buffer of size *nbytes*, and initializes it. Memory for the buffer is allocated from the system memory partition.

RETURNS

The ID of the ring buffer, or **NULL** if memory cannot be allocated.

ERRNO

N/A.

SEE ALSO

rngLib

rngDelete( )

NAME **rngDelete( )** – delete a ring buffer

SYNOPSIS

```
void rngDelete
    (
    FAST RING_ID ringId  /* ring buffer to delete */
    )
```

DESCRIPTION This routine deletes a specified ring buffer. Any data currently in the buffer will be lost.

RETURNS N/A

ERRNO N/A.

SEE ALSO **rngLib**

rngFlush( )

NAME **rngFlush( )** – make a ring buffer empty

SYNOPSIS

```
void rngFlush
    (
    FAST RING_ID ringId  /* ring buffer to initialize */
    )
```

DESCRIPTION This routine initializes a specified ring buffer to be empty. Any data currently in the buffer will be lost.

RETURNS N/A

ERRNO N/A.

SEE ALSO **rngLib**

rngFreeBytes( )

NAME **rngFreeBytes( )** – determine the number of free bytes in a ring buffer

SYNOPSIS

```
int rngFreeBytes
    (
    FAST RING_ID ringId  /* ring buffer to examine */
    )
```

DESCRIPTION This routine determines the number of bytes currently unused in a specified ring buffer.

RETURNS The number of unused bytes in the ring buffer.

ERRNO N/A.

SEE ALSO **rngLib**

rngIsEmpty( )

NAME **rngIsEmpty( )** – test if a ring buffer is empty

SYNOPSIS

```
BOOL rngIsEmpty
    (
    RING_ID ringId  /* ring buffer to test */
    )
```

DESCRIPTION This routine determines if a specified ring buffer is empty.

RETURNS **TRUE** if empty, **FALSE** if not.

ERRNO N/A.

SEE ALSO **rngLib**

rngIsFull( )

NAME **rngIsFull( )** – test if a ring buffer is full (no more room)

SYNOPSIS

```
BOOL rngIsFull
```

```
    (
    FAST RING_ID ringId  /* ring buffer to test */
    )
```

DESCRIPTION This routine determines if a specified ring buffer is completely full.

RETURNS **TRUE** if full, **FALSE** if not.

ERRNO N/A.

SEE ALSO **rngLib**

rngMoveAhead( )

NAME **rngMoveAhead( )** – advance a ring pointer by *n* bytes

SYNOPSIS

```
void rngMoveAhead
    (
    FAST RING_ID ringId,  /* ring buffer to be advanced                  */
    FAST int     n        /* number of bytes ahead to move input pointer */
    )
```

DESCRIPTION This routine advances the ring buffer input pointer by *n* bytes. This makes *n* bytes available in the ring buffer, after having been written ahead in the ring buffer with **rngPutAhead( )**.

RETURNS N/A

ERRNO N/A.

SEE ALSO **rngLib**

rngNBytes( )

NAME **rngNBytes( )** – determine the number of bytes in a ring buffer

SYNOPSIS

```
int rngNBytes
    (
    FAST RING_ID ringId  /* ring buffer to be enumerated */
    )
```

DESCRIPTION This routine determines the number of bytes currently in a specified ring buffer.

RETURNS The number of bytes filled in the ring buffer.

ERRNO N/A.

SEE ALSO **rngLib**

rngPutAhead( )

NAME **rngPutAhead( )** – put a byte ahead in a ring buffer without moving ring pointers

SYNOPSIS

```
void rngPutAhead
    (
    FAST RING_ID ringId,  /* ring buffer to put byte in    */
    char         byte,    /* byte to be put in ring        */
    int          offset   /* offset beyond next input byte where to put byte
*/
    )
```

DESCRIPTION This routine writes a byte into the ring, but does not move the ring buffer pointers. Thus the byte will not yet be available to **rngBufGet( )** calls. The byte is written *offset* bytes ahead of the next input location in the ring. Thus, an offset of 0 puts the byte in the same position as would **RNG_ELEM_PUT** would put a byte, except that the input pointer is not updated.

Bytes written ahead in the ring buffer with this routine can be made available all at once by subsequently moving the ring buffer pointers with the routine **rngMoveAhead( )**.

Before calling **rngPutAhead( )**, the caller must verify that at least *offset* + 1 bytes are available in the ring buffer.

RETURNS N/A

ERRNO N/A.

SEE ALSO **rngLib**

romStart( )

NAME **romStart( )** – generic ROM initialization

SYNOPSIS

```
void romStart
```

```
    (
    FAST int startType  /* start type */
    )
```

DESCRIPTION

This is the first C code executed after reset.

This routine is called by the assembly start-up code in **romInit( )**. It clears memory, copies ROM to RAM, and possibly invokes the uncompressor. It then jumps to the entry point of the uncompressed object code.

RETURNS

N/A

ERRNO

SEE ALSO

bootInit

round( )

NAME

round( ) – round a number to the nearest integer

SYNOPSIS

```
double round
    (
    double x  /* value to round */
    )
```

DESCRIPTION

This routine rounds a double-precision value *x* to the nearest integral value.

RETURNS

The double-precision representation of *x* rounded to the nearest integral value.

ERRNO

Not Available

SEE ALSO

mathALib

roundf( )

NAME

roundf( ) – round a number to the nearest integer

SYNOPSIS

```
float roundf
    (
    float x  /* argument */
    )
```

DESCRIPTION This routine rounds a single-precision value *x* to the nearest integral value.

RETURNS The single-precision representation of *x* rounded to the nearest integral value.

ERRNO Not Available

SEE ALSO **mathALib**

rtgRegister( )

NAME **rtgRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void rtgRegister(void)
```

DESCRIPTION This routine registers the RealTek driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **rtl8169VxbEnd**

rtlRegister( )

NAME **rtlRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void rtlRegister(void)
```

DESCRIPTION This routine registers the RealTek driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **rtl8139VxbEnd**

rtpDelete( )

NAME

rtpDelete( ) – terminates a real time process (RTP)

SYNOPSIS

```
STATUS rtpDelete
    (
    RTP_ID rtpId,     /* ID of the RTP to be deleted */
    int    options,   /* options for deletion */
    int    delStatus  /* exit status */
    )
```

DESCRIPTION

This routine terminates an RTP from the system. The termination of the RTP removes all objects (including tasks) owned by the RTP from the system via the resource reclamation facility. Any mapped memory in the RTP will be unmapped and memory will be freed back to the system. Memory allocated to the RTP for the executable file will also be freed back to 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.

Shared data regions created and mapped in the RTP will be unmapped. If the RTP is the last client reference to the shared data, the termination of the RTP will trigger a deletion of the shared data region. For more information on shared data regions, please refer to **sdLib**.

Shared libraries created and mapped by an RTP will also be unmapped. As with shared data regions, if the terminating RTP is the last reference to the shared library, the termination of the RTP will trigger the deletion of the shared library from the system.

There is currently no user-available values for the *options* parameter. This parameter should always be set to zero.

delStatus is the exit status of the RTP that can be extracted by the parent of this RTP through the **wait( )** system call.

Users may install RTP delete hooks to be called before the termination of the victim RTP's resources. These hooks must reside within the kernel. For hooks that access the RTP's user space, the hooks are responsible for switching into the context of the RTP to perform the access. The delete hooks are called after tasks within the RTP are terminated. To add a delete hook routine, use the following:

rtpDeleteHookAdd( )
Add a hook routine to be called during the termination of an RTP.

WARNING

rtpDelete( ) may not be called from an Interrupt Service Routine (ISR).

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK, or **ERROR** if RTP can not be deleted

ERRNOS

Possible errnos generated are:

S_rtpLib_INVALID_RTP_ID
The specified rtpId parameter is not a valid ID or is in the deleted state already.

S_objLib_OBJ_ID_ERROR
The specified rtpId is invalid for the **rtpDelete( )** operation.

S_objLib_OBJ_DELETED
The RTP object has already been terminated and the RTP no longer exists.

SEE ALSO

rtpLib, **rtpSpawn( )**, **sdDelete( )**

rtpDeleteHookAdd( )

NAME

rtpDeleteHookAdd( ) – add a routine to be called when RTPs are deleted

SYNOPSIS

```
STATUS rtpDeleteHookAdd
    (
    RTP_DELETE_HOOK hook,      /* routine to be called when RTPs are deleted
*/
    BOOL            addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called whenever an RTP is deleted. The hook routine should have the following prototype:

```
    void deleteHook
        (
        const RTP_ID rtpId     /* RTP ID of the RTP about to be deleted */
        const int    exitCode   /* exit code or delete status for the RTP */
        )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (in other words, it will be the first hook to execute). It is typical for delete hooks to be added at the head of the table. Doing so enables the most recently added hooks to execute first. This is useful when hook routines have dependencies among themselves, and the order in which the hooks execute is important.

RTP delete hooks are called from **rtpDelete( )** before any deletion is done. Delete hooks are not expected to return anything (return values if any are not checked).

RETURNS

OK, or **ERROR** if the table of RTP delete routines is full.

ERRNO

N/A.

SEE ALSO **rtpHookLib**, **rtpDeleteHookDelete( )**

rtpDeleteHookDelete( )

NAME **rtpDeleteHookDelete( )** – delete a previously added RTP delete hook routine

SYNOPSIS

```
STATUS rtpDeleteHookDelete
    (
    RTP_DELETE_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each RTP delete.

RETURNS **OK**, or **ERROR** if the routine is not in the table of RTP delete routines.

ERRNO N/A.

SEE ALSO **rtpHookLib**, **rtpDeleteHookAdd( )**

rtpHelp( )

NAME **rtpHelp( )** – print a synopsis of RTP-related shell commands

SYNOPSIS

```
void rtpHelp (void)
```

DESCRIPTION This routine prints the synopsis of the shell commands applying to Real-Time Processes.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrRtpLib**

rtpHookShow( )

NAME

rtpHookShow( ) – display all installed RTP hooks

SYNOPSIS

```
void rtpHookShow (void)
```

DESCRIPTION

This routine displays the contents of all three RTP hook tables, the pre-create, post-create, and delete hook tables.

EXAMPLE

The following example shows a hypothetical set of RTP hook table contents

```
-> rtpHookShow

RTP Pre-Create Hook Table:

hookfunc1
hookfunc2
hookfunc3

RTP Post-Create Hook Table:

hookfunc4
hookfunc5
hookfunc6

RTP Init-Complete Hook Table:

hookfunc7
hookfunc8
hookfunc9

RTP Delete Hook Table:

hookfunc12
hookfunc11
hookfunc10
value = 1 = 0x1
->
```

RETURNS

N/A

ERRNOS

N/A

SEE ALSO

rtpShow, **rtpShow( )**, **hookShow( )**, **syscallHookShow( )**

rtpInfoGet( )

NAME

rtpInfoGet( ) – Get specific information on an RTP

SYNOPSIS

```
STATUS rtpInfoGet
    (
    RTP_ID     rtpId,      /* RTP ID to get info */
    RTP_DESC * rtpStruct  /* Location to store RTP info */
    )
```

DESCRIPTION

This routine obtains information about an RTP and stores the information in the specified RTP descriptor *rtpStruct*. The information stored in the descriptor, for the most part, is a snapshot copy of the information about the RTP object. The descriptor must have been allocated before calling this function, and the memory for it must come from the the calling task's memory space. To allocate the memory for the descriptor from the calling task's memory space, either use **malloc( )** within the calling task or declare the structure as an automatic variable in the calling task, placing it on the calling task's stack.

The rtpStruct structure looks like the following:

```
typedef struct
   {
   char        pathName[VX_RTP_NAME_LENGTH+1]; // pointer to executable path
   int         status;          // the state of the RTP
   UINT32      options;         // option bits, e.g. debug, symtable
   void *      entrAddr;        // entry point of ELF file
   int         initTaskId;      // the initial task ID
   INT32       taskCnt;         // number of tasks in the RTP
   RTP_ID     parentId;        // RTP ID of the parent
   } RTP_DESC;
```

The length of the pathName field is limited to **VX_RTP_NAME_LENGTH** (255). The errno **S_rtpLib_RTP_NAME_MAX** will be set if the RTP's executable pathName exceeds this limit.

The initTaskId will be 0 if the initial task of the RTP was deleted at the time this routine is called. The initTaskId will also be 0 if the caller is a task in a different RTP, as initial task is private to an RTP.

The IDs of the initTaskId and parentId are the WIND kernel IDs, when the routine is used in the kernel. **objShow( )** may be called directly to display information on these objects. If the routine is invoked within an RTP, these IDs are opaque IDs local to the RTP. To display information on these IDs, use the **objHandleShow( )** routine on the opaque IDs.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK or **ERROR**

ERRNOS

S_objLib_OBJ_ID_ERROR
Invalid RTP ID or null *rtpStruct* parameter.

S_rtpLib_RTP_NAME_LENGTH_EXCEEDED
The actual path name of the RTP exceeds the **VX_RTP_NAME_LENGTH** (255).

SEE ALSO

rtpUtilLib, **rtpLib**, **rtpShow( )**

rtpInitCompleteHookAdd( )

NAME

rtpInitCompleteHookAdd( ) – Add routine to be called after RTP init-complete.

SYNOPSIS

```
STATUS rtpInitCompleteHookAdd
    (
    RTP_INIT_COMPLETE_HOOK hook,      /* routine to be called after rtp init 
*/
    BOOL                   addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called after an RTP is created and fully initialized (i.e. just before its initial task starts running). Init-Complete hook routines should have the following prototype:

```
void rtpInitCompleteHook
    (
    const RTP_ID rtpId        /* ID of the created RTP */
    )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

RTP Init-Complete hooks are called as a result of calling **rtpSpawn( )**, and are called after the RTP's creation and initialization are fully complete. Init-Complete hooks are used as a notification point for debuggers and other tools that may want to access the newly created RTP's memory space.

Init-Complete hooks are a notification point, and can't return an **ERROR**. Therefore, they should typically not create any objects or perform any actions that are critical to the existence of the RTP. Consider using a post-create hook if this functionality is required.

RETURNS

OK, or **ERROR** if the table of RTP Init-Complete routines is full.

ERRNO

N/A.

SEE ALSO

rtpHookLib, **rtpInitCompleteHookDelete( )**

rtpInitCompleteHookDelete( )

NAME **rtpInitCompleteHookDelete( )** – delete a previously added RTP init-complete hook

SYNOPSIS

```
STATUS rtpInitCompleteHookDelete
    (
    RTP_INIT_COMPLETE_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of RTP Init-Complete hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **rtpHookLib**, **rtpInitCompleteHookAdd( )**

rtpKill( )

NAME **rtpKill( )** – send a signal to a RTP

SYNOPSIS

```
int rtpKill
    (
    RTP_ID rtpId,
    int    signo
    )
```

DESCRIPTION This routine sends a kill signal *signo* to the RTP specified by *pRtpId*. If signo equals -1, it will use objEach to go through the list of RTP's and send a kill signal to each RTP.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **rtpSigLib**

rtpLkAddr( )

NAME

rtpLkAddr( ) – list symbols in an RTP whose values are near a specified value

SYNOPSIS

```
void rtpLkAddr
    (
    UINT   addr,  /* address around which to look */
    RTP_ID rtpId  /* RTP to look for symbols in */
    )
```

DESCRIPTION

This command lists the symbols in the RTP's symbol table that are near a specified value. The symbols that are displayed include:

- symbols whose values are immediately less than the specified value
- symbols with the specified value
- succeeding symbols, until at least 12 symbols have been displayed

The results of this command depend partly on what options were used to spawn the RTP. The default options cause only global symbols from the ELF file used to spawn the RTP to be available for this routine.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

usrRtpLib, **rtpLib**, **symLib**, **symEach( )**, the VxWorks programmer guides, the IDE and host tools guides.

rtpLkup( )

NAME

rtpLkup( ) – list symbols from an RTP's symbol table

SYNOPSIS

```
void rtpLkup
    (
    char * substr,  /* substring to match */
    RTP_ID rtpId    /* Id of rtp to search for symbol in */
    )
```

DESCRIPTION

This command lists all symbols in the RTP's symbol table whose names contain the string *substr*. If *substr* is omitted or is 0, a short summary of symbol table statistics is printed. If *substr* is the empty string (""), all symbols in the table are listed.

By default, **rtpLkup( )** displays 22 symbols at a time. This can be changed by modifying the global variable **symLkupPgSz**. If this variable is set to 0, **rtpLkup( )** displays all the symbols without interruption.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrRtpLib**, **usrLib**, **rtpLib**, **symLib**, **symEach( )**, the VxWorks programmer guides, the IDE and host tools programer guides.

rtpMemShow( )

NAME **rtpMemShow( )** – display memory context information for real time proceses

SYNOPSIS

```
STATUS rtpMemShow
    (
    char * rtpNameOrId,  /* RTP name or ID */
    int    level         /* detail level */
    )
```

DESCRIPTION This routine displays memory context information for a real time process. This routine takes two parameters, *rtpNameOrId* and *level*. The first parameter can either be an RTP ID or an RTP name string. The second parameter is the level of detail to display the information for the RTPs. If the *level* is 0, then it displays a summary memory context information for the specified RTP. If the *level* is 1, then **rtpMemShow( )** displays detailed memory context information. If the *level* is 2, then **rtpMemShow( )** also displays POSIX mapped file and mapped object information.

SUMMARY INFORMATION EXAMPLE

The following example shows the summary output for all RTPs in the system.

C-interpreter shell:

```
-> rtpMemShow 0x27ba010

Memory Information for 0x27bb000 RTP (name = "myRtp.vxe"):

Virtual Memory Context:
=======================

Virtual Memory Context ID:          0x301ecd0
Private Virtual Memory Allocated:   0x6a000 bytes
Private Virtual Memory Mapped:      0x66000 bytes

value = 0 = 0x0
```

DETAILED INFORMATION EXAMPLE

The following example shows the detailed output for an RTP.

C-interpreter shell:

```
-> rtpMemShow 1635018888, 1

Memory Information for 0x61746888 RTP (name = "< in/tmMmanFdLib.vxe"):

Virtual Memory Context:
=======================

Virtual Memory Context ID:          0x606ad290
Private Virtual Memory Allocated:   0x6a000 bytes
Private Virtual Memory Mapped:      0x66000 bytes

Private Mappings:

VIRTUAL ADDR  BLOCK LENGTH  PHYSICAL ADDR  PROT (S/U)  CACHE     SPECIAL
------------  ------------  -------------  ----------  -------   ------------
0x63000000    0x00012000    0x6174a000     R-X / R-X   CB-/--/-  --
0x63012000    0x00010000    0x6175c000     RWX / RWX   CB-/--/-  --
0x63022000    0x00002000    ***unmapped***
0x63024000    0x00010000    0x6176c000     RWX / RWX   CB-/--/-  --
0x63034000    0x00002000    ***unmapped***
0x63036000    0x00032000    0x6177c000     RWX / RWX   CB-/--/-  --
0x63068000    0x00002000    0x617ae000     R-X / R-X   CB-/--/-  --

Shared Data Mappings:

VIRTUAL ADDR  BLOCK LENGTH  PHYSICAL ADDR  PROT (S/U)  CACHE     SPECIAL
------------  ------------  -------------  ----------  -------   ------------
0x6306c000    0x00002000    0x617b8000     RWX / RWX   CB-/--/-  --

Shared Object (POSIX) Mappings:

VIRTUAL ADDR  BLOCK LENGTH  PHYSICAL ADDR  PROT (S/U)  CACHE     SPECIAL
------------  ------------  -------------  ----------  -------   ------------
0x6306a000    0x00002000    0x617b0000     R-X / R-X   CB-/--/-  --

value = 0 = 0x0
```

For further information on the fields from the Memory Context section, see **vxContextShow( )**.

The private mappings of an RTP consist of the memory mapped objects (mapped with **MAP_PRIVATE**), the RTP's code segments (text/data/bss), the RTP's heap, and the stacks of the tasks running in the RTP.

The Shared Data Mappings consists of Shared Data regions opened with the **sdLib** API; this also includes shared library text segments.

The Shared Object Mappings consist of mappings obtained with **mmap( )** using the **MAP_SHARED** flag.

MAPPED OBJECT INFORMATION EXAMPLE

The following example shows POSIX memory mapped files information.

C-interpreter shell:

```
-> rtpMemShow 1635018888, 2

[removed detailed memory context information, same as above]

Memory Mapped Objects (POSIX):
==============================

 ADDRESS    LENGTH     PROT  FLAGS    OFFSET             OBJECT
---------- ---------- ---- -------- ------------------ ----------------------
0x63036000 0x00010000  RW-  PRIVATE N/A                ***anonymous***
0x63046000 0x00010000  RW-  PRIVATE N/A                ***anonymous***
0x63056000 0x00012000  RW-  PRIVATE N/A                ***anonymous***
0x63068000 0x00002000  R--  PRIVATE 0x0000000000000000 /pxFs/mmapFd1
0x6306a000 0x00002000  R--  SHARED  0x0000000000000000 /pxFs/mmapFd2
value = 0 = 0x0
```

Note that the PROT value shown in the Memory Mapped Objects section is the parameter passed to **mmap( )** (i.e. the bit values of **PROT_READ**, **PROT_WRITE**, **PROT_EXE** passed via the *prot* parameter). The PROT value displayed in the Memory Context section shows the actual protection that resulted after setting the corresponding MMU protection attributes. Depending on the constraints of the processor architecture and system configuration these may or may not be the same.

COMMAND INTERPRETER

For the command-interpreter shell, use the **rtp meminfo** command.

RETURNS

OK if success, **ERROR** otherwise

ERRNOS

Possible errnos generated by this function include:

S_objLib_OBJ_ID_ERROR
An incorrect RTP ID was provided.

S_objLib_ACCESS_DENIED
Unable to get exclusive access to the RTP or the RTP list.

SEE ALSO

rtpShow, **rtpShow( )**, **rtpLib**, **rtpUtilLib**, **vmContextShow( )**, the VxWorks programmer guides.

rtpPostCreateHookAdd( )

NAME

rtpPostCreateHookAdd( ) – add a routine to be called just after RTP creation.

SYNOPSIS

```
STATUS rtpPostCreateHookAdd
    (
    RTP_POST_CREATE_HOOK hook,      /* routine to be called on rtp creation
*/
    BOOL                 addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called just after an RTP and its initial task are created, but before the newly created RTP starts running.

Upon creation, all routines specified by **rtpPostCreateHookAdd( )** will be called in the context of the creating RTP, so any objects created by an post-create hook will be owned by the caller's RTP rather than the newly created RTP. To set the ownership of newly created objects to the new RTP, **objOwnerSet( )** should be used. For example:

```
objOwnerSet (createdObjId, rtpId)
```

Post-create hook routines should have the following prototype:

```
    STATUS rtpPostCreateHook
        (
        const RTP_ID rtpId        /* ID of the created RTP */
        )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

RTP Post-creation hooks are called from **rtpSpawn( )**, and should return either **OK** or **ERROR**. If the return value from a post-create hook is anything other than **OK**, the created RTP and its initial task are deleted, and **rtpSpawn( )** returns **ERROR**. Post-creation hooks can be used to perform additional application-specific resource allocation etc where the created RTP's details should be known. Should such allocations fail, users have the option of reversing RTP creation by returning **ERROR**.

RETURNS

OK, or **ERROR** if the table of RTP post-create routines is full.

ERRNO

N/A.

SEE ALSO

rtpHookLib, **rtpPostCreateHookDelete( )**

rtpPostCreateHookDelete( )

NAME

rtpPostCreateHookDelete( ) – delete a previously added RTP post-create hook.

SYNOPSIS

```
STATUS rtpPostCreateHookDelete
    (
    RTP_POST_CREATE_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION

This routine removes a specified hook routine from the list of RTP post-create hook routines.

RETURNS

OK on success, or **ERROR** if the hook routine was not found.

ERRNO

S_hookLib_HOOK_NOT_FOUND

SEE ALSO

rtpHookLib, **rtpPostCreateHookAdd( )**

rtpPreCreateHookAdd( )

NAME

rtpPreCreateHookAdd( ) – add a routine to be called before RTP creation.

SYNOPSIS

```
STATUS rtpPreCreateHookAdd
    (
    RTP_PRE_CREATE_HOOK hook,      /* routine to be called on rtp creation */
    BOOL                addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called just before an RTP is created. The hook routine should have the following prototype:

```
    STATUS rtpPreCreateHook
        (
        const char * rtpFileName, /* Null-terminated path to executable */
        const char * argv[],      /* pointer to NULL terminated argv array */
        const char * envp[],      /* pointer to NULL terminated envp array */
        int          priority,    /* priority of initial task */
        int          uStackSize,  /* User space stack size for intial task */
        int          options,     /* the options passed to RTP */
        int          taskOptions  /* Task options for RTP's initial task */
        )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

RTP Pre-creation hooks are called from **rtpSpawn( )** before RTP creation begins. Hooks should return either **OK** or **ERROR**. If the return value from any hook is anything other than **OK**, RTP creation does not proceed. Pre-creation hooks can be used to implement rudimentary authentication schemes by rejecting RTP spawn requests before any action is taken.

RETURNS **OK**, or **ERROR** if the table of RTP pre-create routines is full.

ERRNO N/A.

SEE ALSO **rtpHookLib**, **rtpPreCreateHookDelete( )**

rtpPreCreateHookDelete( )

NAME **rtpPreCreateHookDelete( )** – delete a previously added RTP pre-create hook.

SYNOPSIS

```
STATUS rtpPreCreateHookDelete
    (
    RTP_PRE_CREATE_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of RTP pre- create hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **rtpHookLib**, **rtpPreCreateHookAdd( )**

rtpShlShow( )

NAME **rtpShlShow( )** – Display shared library information for an RTP

SYNOPSIS

```
STATUS rtpShlShow
    (
    RTP_ID rtpId  /* RTP to display SHLs on */
    )
```

DESCRIPTION

This routine takes an RTP as parameter and displays SHL information of all SHLs for the RTP. The **SHL_ID** information displayed by this routine can be used by **shlShow( )** to display detail information for a specific SHL.

The *options* field may be specified to not print the header for the SHLs displayed.

- **SHL_SHOW_NO_HDR** (0x0001) Do not display the header for this routine.

EXAMPLE

Below is an example display of a shared library.

```
-> rtpShlShow 0x8d1ea9c
    SHL NAME              ID       TEXT_ADDR  TEXT_SIZE  DATA_ADDR  DATA_SIZE
-------------------- ---------- ---------- ---------- ---------- ----------
< tty/slDfw/libSo.so          1 0xff435000      0x574 0xff46d000      0x628
```

RETURNS

OK, or **ERROR** if invalid RTP id

ERRNOS

Possible errnos generated by this function include:

S_objLib_OBJ_ID_ERROR
An incorrect SHL ID was provided.

S_objLib_ACCESS_DENIED
Unable to get exclusive access to the SHL list.

SEE ALSO

shlShow, **shlShow( )**, **rtpShow( )**

rtpShow( )

NAME

rtpShow( ) – display information for real time proceses

SYNOPSIS

```
BOOL rtpShow
    (
    char * rtpNameOrId,  /* RTP name or ID */
    int    level         /* 0 = summary, 1 = detailed, 2 = all in details */
    )
```

DESCRIPTION

This routine displays information for a real time process. This routine takes two parameters, *rtpNameOrId* and *level*. The first parameter can either be an RTP ID or an RTP name string. The second parameter is the level of detail to display the information for the RTPs.

Depending on the level and the RTP ID specified, the information displayed differs. If the *level* is 0, then it displays the summary information for either the specified RTP or all RTPs in the system. If the *level* is 1, then **rtpShow( )** displays the detailed information, for the specified RTP or the current task's home RTP (if RTP ID is **NULL**, and called from the command shell). If *level* is 2, **rtpShow( )** displays the detailed information for all RTPs in the system, regardless of the RTP ID you specify. Refer to the table for more information.

Level	RTP Name or ID	Meaning	Cmd Shell Equivalent
0	0	Display summary information for all RTPs.	**rtp**
0	RTP	Display summary information for specified RTP.	**rtp** *rtpId*
1	RTP	Display detailed information for specified RTP.	**rtp info** *rtpId*
2	ANY	Display detailed information for all RTP.	**rtp info**

In summary mode, **rtpShow( )** only displays the RTP name (including the path) up to a maximum of 20 characters long. If the name is more than 20 characters, it will be truncated to 20 characters for displaying purposes. Preceding the truncated name, a "< " will be displayed to indicate that the name is more than 20 characters long. To get a display of the full RTP name, display the RTP with the *level* set to 1.

SUMMARY INFORMATION EXAMPLE

The following example shows the summary output for all RTPs in the system. If a RTP ID (or name) is specified, only the information for that RTP will be displayed.

C-interpreter shell:

```
-> rtpShow

      NAME              ID          STATE     ENTRY ADDR  OPTIONS  TASK CNT
------------------- ---------- ------------- ---------- --------- --------
< /apps/myApp.vxe     0x4a9450 STATE_NORMAL  0xa0000148      0x11        1

value = 1 = 0x1
```

For the command-interpreter shell, use the command **rtp**.

The display contains the following fields:

Field	Meaning
NAME	The name of the RTP, using the executable filename.
ID	The numeric ID associated with the RTP.
STATUS	State of the RTP. Refer to the table below for more information.
ENTRY ADRS	The entry routine address of the application.
OPTIONS	Options specified for the RTP. Refer to the options below.
TASK CNT	Number of tasks in the RTP.

STATE_CREATE
The RTP is currently in its create phase.

STATE_CREATE+S
The RTP is in **RTP_STATE_CREATE** state and its status is in **RTP_STATUS_STOP**. This indicates that the RTP currently in the create phase but is suspended during this phase.

STATE_CREATE+D
The RTP is in **RTP_STATE_CREATE** state and its status is in **RTP_STATUS_ELECTED_DELETER**. This indicates that the RTP has encountered an error during its create phase and has started its deletion.

STATE_NORMAL
The RTP has completed initializing and its currently running normally.

STATE_NORMAL+S
The RTP is in **RTP_STATE_NORMAL** state and its status is in **RTP_STATUS_STOP**. This indicates that the RTP currently has all its task in **TASK_STOP** state.

STATE_NORMAL+D
The RTP is in **RTP_STATE_NORMAL** state and its status is in **RTP_STATUS_ELECTED_DELETER**. This indicates that the RTP has initiated its delete phase.

STATE_DELETE
The RTP is in the processing of terminating and cleaning up.

STATE_DELETE+S
The RTP is in **RTP_STATE_DELETE** state and its status is in **RTP_STATUS_STOP**. This indicates that the RTP is in the delete phase but tasks in the RTP has been stopped.

STATE_DELETE+D
The RTP is in **RTP_STATE_DELETE** state and its status is **RTP_STATUS_ELECTED_DELETER**. This indicates that the RTP has initiated its delete phase. The deletion process can not be undone.

DETAILED INFORMATION EXAMPLE

The following example shows the detailed output for a single RTP (i.e. the level was specified as 1). If the level is specified as 2, the detailed information is displayed for all RTP and the user is prompted to press **return** or **Q** between each RTP.

C-interpreter shell:

```
-> rtpShow 0x4a9450, 1

       NAME              ID          STATE    ENTRY ADDR  OPTIONS  TASK CNT
------------------ ---------- ------------- ---------- --------- --------
< /apps/myApp.vxe    0x4a9450 STATE_NORMAL  0xa0000148      0x11        1

Full Name:     /usr/apps/myApp.vxe
Options:       (      0x11) RTP_GLOBAL_SYMBOLS RTP_DEBUG
rtpId->pArgv ptr:  0xa001ef8c
rtpId->pEnv ptr:   0xa001ef94
Initial Task ID:   0x4ab020
Symbol table:      0x47b6c0

SEGMENT START ADDR    SIZE
------- ---------- ----------
text    0xa0000080      49068
data    0xa000d02c       1424
```

```
bss     0xa000d5bc      15310

Shared Libraries:
-----------------
     SHL NAME           ID      TEXT_ADDR  TEXT_SIZE  DATA_ADDR  DATA_SIZE
------------------- --------- ---------- ---------- ---------- ----------
< tty/sls/libSo.so          1 0xff435000      0x574 0xff46d000      0x628

Type <CR> to continue, Q<CR> to stop:
value = 1 = 0x1
```

For the command-interpreter shell, use the command **rtp info**.

The summary line contains the same fields as explained above. The additional information is explained in the following table:

Field	Meaning
Full Name	The complete name for the RTP.
Options	Detailed breakdown of the options word (see **rtpCreate( )**).
rtpId->pArgv ptr	The address in user space where the arguments are stored.
rtpId->pEnv ptr	Address in user space of the environment string.
SEGMENT Information	The text and data information for the RTP executable.
Initial Task ID	The Initial task ID of the RTP.
Symbol table	The ID of the symbol table holding this RTP's symbols.
Shared Libraries	The summary list of shared libraries used by the RTP.

RETURNS

TRUE if success, **FALSE** otherwise

ERRNOS

Possible errnos generated by this function include:

S_objLib_OBJ_ID_ERROR
An incorrect RTP ID was provided.

S_objLib_ACCESS_DENIED
Unable to get exclusive access to the RTP or the RTP list.

SEE ALSO

rtpShow, **rtpMemShow( )**, **rtpLib**, **rtpUtilLib**, **vmContextShow( )**, the VxWorks programmer guides.

rtpSigqueue( )

NAME

rtpSigqueue( ) – send a queued signal to a RTP

SYNOPSIS

```
int rtpSigqueue
    (
    RTP_ID              rtpId,
    int                 signo,
    const union sigval value
    )
```

DESCRIPTION The function **sigqueue( )** sends the queued signal specified by *signo* with the signal-parameter value specified by *value* to the process specified by *pRtpId*.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**
EAGAIN

SEE ALSO **rtpSigLib**

rtpSp( )

NAME **rtpSp( )** – launch a RTP with default options.

SYNOPSIS

```
int rtpSp
    (
    char * execAndArgs,       /* path to the executable file + arguments */
    int    initTaskPrio,      /* priority of RTP's initial task */
    int    userStackSize,     /* size of the initial task's user stack */
    int    launchOptions,     /* options to apply to the RTP at launch */
    int    launchTaskOptions  /* task option for the RTP's initial task */
    )
```

DESCRIPTION This command is a short form of the underlying **rtpSpawn( )** routine, convenient for launching Real-Time Process (RTP) from the shell.

The executable file used to launch the application as well as the arguments to be passed to the **main( )** routine of the application are all contained in the routine's first argument: the string *execAndArgs*. For instance:

```
rtpSp "/folk/me/myVxWorksApp.vxe firstArgument secondArgument"
```

The space character is interpreted as the separator between each element of the string. Any application argument can be a sub-string of its own providing that it is surrounded with escaped double quote characters. For instance:

```
rtpSp "/folk/me/myVxWorksApp.vxe \"first argument\" \"second argument\""
```

Although it is possible to specify properties and options to apply to the RTP's initial task using the parameters *initTaskPrio*, *userStackSize*, *launchOptions* and *launchTaskOptions* these parameters may be left unspecified (i.e. left null). In this case, default values for these properties and options will be applied. These default values may be overriden by updating the values of the following global variables:

rtpSpPriority:
    220 - initial task's priority.

rtpSpStackSize:
    65,535 - initial task's user stack size, in bytes.

rtpSpOptions:
    0x1 (**RTP_GLOBAL_SYMBOLS**) - OR'ed options to apply to the RTP, see **rtpSpawn( )** for details.

If you to want to spawn an RTP with no option bits set but do not want to modify the value of the rtpSpOptions variable, then rtpSp should be called with *launchOptions* set to -1. The special value -1 forces the default value in rtpSpOptions to be ignored. This results in the RTP to be started without any option set (the same as if the rtpSpOptions variable held a null value).

rtpSpTaskOptions:
    VX_FP_TASK - OR'ed options to apply to the initial task of the RTP, see also **taskSpawn( )**.

As in the case of *launchOptions*, if *rtpSpTaskOptions* is set to -1, the default value in rtpSpTaskOptions is ignored and the initial task of the RTP is spawned with no task option bits (i.e. no floating-point support etc).

Additionally, the delay to wait before terminating the execution of the command after an application has been launched can be contolled using:

rtpSpDelay:
    100 - number of ticks to wait.

Note that these global variables also apply to the command interpreter mode of the shell.

CAVEAT the content of the string *execAndArgs* is modified by this routine when parsed.

RETURNS A **RTP_ID** on success, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **usrRtpLib**, **rtpLib**, **rtpSpawn( )**, **taskSpawn( )**

rtpSpawn( )

NAME **rtpSpawn( )** – spawns a new Real Time Process (RTP) in the system

SYNOPSIS

```
RTP_ID rtpSpawn
```

```
(
const char * rtpFileName,  /* Null terminated path to executable */
const char * argv[],       /* Pointer to NULL terminated argv array */
const char * envp[],       /* Pointer to NULL terminated envp array */
int          priority,     /* Priority of initial task */
int          uStackSize,   /* User space stack size for initial task */
int          options,      /* The options passed to the RTP */
int          taskOptions   /* Task options for the RTP's initial task */
)
```

DESCRIPTION

This routine creates and initializes a Real Time Process (RTP) in the system, with the specified file as the executable for the RTP.

Each RTP is named. The name is based on the specified executable filename, via the *rtpFileName* argument, loaded in the RTP. This executable file must reside in a filesystem. The filesystem may be external or media-less and bundled (ROMFS) into the VxWorks system.

The first element to the argv[] array, by convention, should be the filename path of the executable. **rtpSpawn( )** does not automatically populate argv[0] to be the executable pathname; the user must set it. Not providing argv[0] with the executable pathname may cause unexpected results if dynamic shared libraries are involved. Below is an example:

```
char * argv[] = {"/usr/test.vxe", NULL};
rtpSpawn (argv[0], argv, NULL, 100, 0x10000, 0, 0);
```

An RTP is a container for resources of the RTP application. Resources that may be associated with an RTP are: tasks, heap memory, and objects. Memory allocated for an RTP is unique in the system. Memory allocated to an RTP are task stacks, heap memory to be used by the user level heap manager, and memory allocated for the text and data segments of the application.

RTPs provide symbol name isolation. An executable may be spawned more than once in the system and the execution of the applications will not interfer with each other.

Tasks in an RTP are scheduled as part of the global scheduling scheme in the system. RTPs are not schedulable entities; only tasks within the RTPs are schedulable. Thus, for an RTP to exist, tasks must exist in it.

The *envp* environment array may be used to pass specific RTP environment variable settings to the application. Environment variables, such as **LD_LIBRARY_PATH**, may be set for an RTP. To obtain environment information for an RTP, use the **getenv( )** routine or the **extern char **environ** variable in the application. Other reserved environment variables can be used to pass information used by the RTP when it initializes:

HEAP_INITIAL_SIZE
Set the initial size of the RTP's heap to a value other than the default (0x10000).

HEAP_MAX_SIZE
Set the maximum size that the RTP's heap may grow to.

HEAP_INCR_SIZE
Set the growth increment when it should be different from the default (a virtual memory page size).

See the application-side **memLib** documentation for more details. Such variables can be used as follows:

```
  char * argv[] = {"/usr/test.vxe", NULL};
  char * envp[] = {"HEAP_INITIAL_SIZE=0x20000", "HEAP_MAX_SIZE=0x100000",
NULL);
  rtpSpawn (argv[0], argv, envp, 100, 0x10000, 0, 0);
```

The creation and initialization of an RTP also creates the initial task of the RTP. This initial task initializes the VxWorks user level library, libc support or **taskLib** support, of the RTP. Three of **rtpSpawn( )**'s parameters are dedicated to setting the initial task's priority, user-side stack and options:

priority:this parameter sets the priority of the RTP's initial task and care should be taken in setting a priority appropriate for an application (i.e. do not leave this parameter set to zero as this would create an initial task of the highest priority in VxWorks, possibly disturbing the functioning the rest of the system. A value between 200 and 220 is usually adequate).

uStackSize:this parameter sets the size of the initial task's user-side stack. If this parameter is left null this size is set to the default value (0x4000 bytes).

taskOptions:this parameter allows to pass options to the initial task created with the RTP. The *taskOptions* parameter has exactly the same value and meaning as the options parameter passed to **taskSpawn( )**. Some task options available for kernel tasks are prohibited for RTP tasks, and will be ignored if set. These are the **VX_SUPERVISOR_MODE** and **VX_UNBREAKABLE** options. The initial task of every RTP is created with the **VX_DEALLOC_STACK** option.

Options may be passed to the **rtpSpawn( )** API to specify the behavior of the RTP:

RTP_GLOBAL_SYMBOLS (0x01)
The global symbols of the executable file will be registered in the RTP's symbol table. This is required when debugging using the embedded debugging facility.

RTP_ALL_SYMBOLS (0x03)
Both the global and local symbols of the executable file will be registered in the RTP's symbol table. This can be helpful when debugging using the embedded debugging facility.

RTP_DEBUG (0x10)
The execution of the RTP will be stopped at startup in order to enable debugging the application.

RTP_BUFFER_VAL_OFF (0x20)
User buffer passed to system calls will not be validated for this RTP. This will reduce the system call overhead, to the detriment of security. This option should be used only once the application code was properly debugged.

RTP_LOADED_WAIT (0x40)
rtpSpawn( ) will not return until the RTP has been instantiated, all code loaded, the RTP's state is **RTP_STATE_NORMAL**, and execution is about to transfer to user mode.

RTP_CPU_AFFINITY_NONE (0x80)
By default the RTP's initial task inherits the CPU affinity of the task that spawned the RTP. This option removes any CPU affinity that would have applied to the initial task (i.e. this task will migrate from one CPU to another). Applies to SMP only.

A set of hooks are provided for users to extend the capabilities of the **rtpSpawn( )** routine. Hooks may be added prior to the creation of the RTP, after the creation of the RTP object and VM context, and also after the loading of the RTP executable file. To add hooks at each of these points, use the following routines:

rtpPreCreateHookAdd( )
Add a hook to be called prior to the creation of the RTP. The hook is executed in the caller's context. This hook facility is useful for validations prior to RTP creation, such as restricting the creation of RTPs to certain RTPs. Note, when this hook fails, RTP delete hook is not called.

rtpPostCreateHookAdd( )
Add a hook to be called after the creation of the RTP object and VM context. Hooks are executed in the context of the caller. Any objects created by the hooks are owned by the caller unless the owner is reset via the **objOwnerSet( )** API. This hook should not attempt to delete the RTP.

The post create hook is useful for extending the initialization of the RTP object such as initializing any user defined structures or objects.

An error returned from the registered hook routines will invoke a termination of the RTP. The RTP state is **RTP_STATE_CREATE**. Please refer to the routine **rtpPostCreateHookAdd( )** for more information.

rtpInitCompleteHookAdd( )
Add a hook to be called after loading of the RTP executable file and before the RTP's initial task starts executing in user mode. The RTP state is **RTP_STATE_NORMAL**.

rtpDeleteHookAdd( )
Add a hook to be called when an RTP is terminated. The hooks are called at the beginning of the **rtpDelete( )** routine. The RTP state is **RTP_STATE_NORMAL**

For more detailed information on RTP hooks, refer to **rtpHookLib**.

The default behavior when an RTP task encounters an error, such as an exception, is that the system will terminate the faulty RTP. However, for debugging purposes, the system may be configured to behave in a **lab** mode where an exception would not terminate the RTP. Instead the faulty task and RTP will be suspended for debugging. To turn on the lab mode refer to the **edrLib** documentation.

WARNING **rtpSpawn( )** may not be called from an Interrupt Service Routine (ISR).

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **RTP_ID** of the new RTP, or **ERROR** otherwise.

ERRNOS Possible errnos returned from this routine are:

S_rtpLib_INVALID_FILE
The path to the executable file is not valid. The *rtpFileName* parameter is either null or the executable file cannot be found via the provided path. A valid path is a path that can be successfully accessed via the kernel shell.

S_rtpLib_INVALID_TASK_OPTION
One or more of the options specified for the initial task are not supported for a user-mode task.

S_rtpLib_INSTANTIATE_FAILED
The RTP object was created but failed to load and reach **RTP_STATE_NORMAL**

SEE ALSO **rtpLib**, **rtpDelete( )**, **rtpInfoGet( )**, **rtpHookLib**, **memLib**, the VxWorks programmer guides.

rtpSymTblIdGet( )

NAME **rtpSymTblIdGet( )** – Get the symbol table ID of an RTP

SYNOPSIS

```
SYMTAB_ID rtpSymTblIdGet
    (
    RTP_ID rtpId  /* RTP ID whose symbol table ID is needed */
    )
```

DESCRIPTION This routine gets and returns the symbol table ID for an RTP's symbol table.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The RTP's symbol table ID or **NULL** if the ID could not be retrieved

ERRNOS **S_objLib_OBJ_ID_ERROR**
RTP object is not valid.

SEE ALSO **rtpUtilLib**, **rtpLib**

rtpSymsAdd( )

NAME

rtpSymsAdd( ) – add symbols from an executable file to a RTP symbol table

SYNOPSIS

```
STATUS rtpSymsAdd
    (
    RTP_ID rtpId,       /* RTP the symbols should be added to */
    UINT32 regPolicy,   /* symbol registration policy */
    char * filePath     /* path and name of the executable file */
    )
```

DESCRIPTION

This command is provided as a help in case a RTP needs to be debugged but has been launched with an empty symbol table. It forces the registration of the symbols from an executable file into a RTP symbol table.

Note that this command does not verify whether the symbols are already in the symbol table and does not prevent the creation of multiple occurences of these symbols.

It is important to understand that symbols are added to the symbol table in the order of their registration and that the most recent entry will hide symbols of same name already registered. The **rtpLkup( )** command will show all occurences of the symbols of a given name so it is possible to use their addresses instead of their names if there is a risk of confusion.

The only required information is the RTP ID (*rtpId* parameter).

The *regPolicy* parameter sets the symbol registration policy. The policy can be one of the following:

0x01 (**RTP_GLOBAL_SYMBOLS**)
Add only global symbols to the symbol table. This is the default when the parameter is left null.

0x02 (**RTP_LOCAL_SYMBOLS**)
Add only local symbols to the symbol table.

0x03 (**RTP_ALL_SYMBOLS**)
Add both local and global symbols to the symbol table.

The *filePath* parameter overrides the path recorded for the RTP. It may be left null if the symbols should be read from the same file as the one used to start the RTP with. This parameter must be used when the symbols should be read from a file stored in a different location than what was recorded when the RTP has been launched. Note that there is no runtime verification that the file corresponds to the executable used to launch the RTP.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK if the symbols could be read and recorded, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **usrRtpLib**, **rtpSymsRemove( )**, **shlSymsAdd( )**, **shlSymsRemove( )**, **rtpSymsForce( )**

rtpSymsOverride( )

NAME **rtpSymsOverride( )** – override the RTP symbol registration policy

SYNOPSIS

```
STATUS rtpSymsOverride
    (
    int overridePolicy  /* override the symbol registration policy */
    )
```

DESCRIPTION This command is provided as a help for debugging and monitoring RTPs launched by other applications. It allows to temporarily bypass the symbol registration policy encoded in RTPs and forces an alternate symbol registration policy which will apply to all applications launched after the change is made. The change is in effect until cancelled.

The *overridePolicy* parameter sets the alternate symbol registration policy. The policy can be one of the following:

-	Prevent any symbol to be registered in the RTP's symbol table.
1	Add only global symbols to the symbol table.
3	Add both local and global symbols to the symbol table.
-1	Cancel any override of the symbol registration policy.

After the override of the RTP symbol registration policy is canceled, any newly launched application will get applied the symbol registration policy set by the parent application, or the shell.

RETURNS **OK** if the override policy could be set, **ERROR** otherwise.

ERRNO Not Available

SEE ALSO **usrRtpLib**, **rtpSymsAdd( )**, **rtpSymsRemove( )**, **shlSymsAdd( )**, **shlSymsRemove( )**

rtpSymsRemove( )

NAME

rtpSymsRemove( ) – remove symbols from a RTP symbol table

SYNOPSIS

```
STATUS rtpSymsRemove
    (
    RTP_ID rtpId,     /* RTP the symbols should be removed from */
    UINT32 remPolicy  /* symbol removal policy */
    )
```

DESCRIPTION

This command forces the removal of symbols from a RTP symbol table.

The *remPolicy* parameter sets the symbol removal policy. The policy can be one of the following:

0x02 (**RTP_LOCAL_SYMBOLS**)
Remove only local symbols from the symbol table.

0x03 (**RTP_ALL_SYMBOLS**)
Removes both local and global symbols from the symbol table.

Note: in the current implementation, this command will also remove the symbols related to shared libraries bound to the RTP. In order to remove only the symbols related to a specific shared library use **shlSymsRemove( )**.

RETURNS

OK if the symbols could be removed, **ERROR** otherwise.

ERRNO

N/A

SEE ALSO

usrRtpLib, **rtpSymsAdd( )**, **shlSymsAdd( )**, **shlSymsRemove( )**, **rtpSymsForce( )**

rtpTaskKill( )

NAME

rtpTaskKill( ) – send a signal to a task

SYNOPSIS

```
int rtpTaskKill
    (
    TASK_ID tid,   /* task to send signal to */
    int     signo  /* signal to send to task */
    )
```

DESCRIPTION

This routine sends a kill signal *signo* to the RTP task specified by *tid*.

RETURNS

OK (0), or **ERROR** (-1) if the task ID or signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **rtpSigLib**

rtpTaskSigqueue( )

NAME **rtpTaskSigqueue( )** – send a queued signal to a task

SYNOPSIS

```
int rtpTaskSigqueue
    (
    TASK_ID            tid,
    int                signo,
    const union sigval value
    )
```

DESCRIPTION The function **rtpTaskSigqueue( )** sends the queued signal specified by *signo* with the signal-parameter value specified by *value* to the RTP task specified by *tid*.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**
EAGAIN

SEE ALSO **rtpSigLib**

rtpi( )

NAME **rtpi( )** – display all tasks within an RTP

SYNOPSIS

```
STATUS rtpi
    (
    RTP_ID rtpId  /* RTP identifier value, or 0 for task's RTP */
    )
```

DESCRIPTION This command displays summary task information on tasks associated with the RTP *rtpId*. For more detailed information, the routine **ti( )** or **taskShow( )** should be used with a specified **TASK_ID**.

This routine displays tasks of an RTP in sorted order if the number of tasks is less than 100. If more than 100 tasks are within an RTP, the tasks will be displayed in the order they are created.

RETURNS

OK, or **ERROR** if *rtpId* is invalid

ERRNO

Possible errno values are:

S_objLib_OBJ_ID_ERROR
An invalid RTP ID was provided.

S_objLib_ACCESS_DENIED
Unable to get exclusive access to the RTP to display tasks.

SEE ALSO

usrRtpLib, **i( )**, **ti( )**, **taskShow( )**

s( )

NAME

s( ) – single-step a task

SYNOPSIS

```
STATUS s
    (
    int      taskNameOrId,  /* task to step; 0 = default        */
    INSTR * addr,           /* address to step to; 0 = next instruction */
    INSTR * addr1           /* address for npc, 0 = next instruction */
    )
```

DESCRIPTION

This routine single-steps a task that is stopped at a breakpoint.

To execute, enter:

```
    -> s [task[,addr[,addr1]]]
```

If *task* is omitted or zero, the last task referenced is assumed. If *addr* is non-zero, then the program counter is changed to *addr*; if *addr1* is non-zero, the next program counter is changed to *addr1*, and the task is stepped.

CAVEAT

When a task is continued, **s( )** does not distinguish between a stopped task or a task stopped by the debugger. Therefore, its use should be restricted to only those tasks being debugged.

NOTE

The next program counter, *addr1*, is currently supported only by SPARC.

RETURNS

OK, or **ERROR** if the debugging package is not installed, the task cannot be found, or the task is not suspended.

ERRNO

N/A

SEE ALSO **dbgLib**, **so( )**, **c( )**, *VxWorks Kernel Programmer's Guide: Kernel Shell*, *VxWorks Command-Line Tools User's Guide 2.2: Host Shell*

salCall( )

NAME **salCall( )** – invoke a socket-based server

SYNOPSIS

```
int salCall
    (
    int    sockfd,    /* client socket fd */
    void * pSendBuf,  /* message buffer */
    int    sendLen,   /* size of message buffer */
    void * pRecvBuf,  /* reply buffer */
    int    recvLen    /* size of reply buffer */
    )
```

DESCRIPTION This routine sends a message to the server associated with the socket descriptor *sockfd* and waits for a reply. The message consists of the *sendLen* bytes pointed at by *pSendBuf*. The reply is placed in the *recvLen* bytes pointed at by *pRecvBuf*. If fewer than *recvLen* bytes are received the unused portion of *pRecvBuf* is not altered; if more than *recvLen* bytes are received the unused portion of the reply may be kept or discarded depending on the socket protocol being used.

If the socket descriptor is used by multiple clients, mutual exclusion needs to be provided before this routines is called. This is to avoid the case when a reply is intercepted by a higher priority task sharing the same *sockfd*.

RETURNS # of bytes placed in reply buffer, for connection based transport, 0 bytes may returned when the called end closes the connection; **ERROR** otherwise.

ERRNO **S_salLib_INVALID_ARGUMENT**
An invalid argument was passed to this routine.

SEE ALSO **salClient**

salCreate( )

NAME **salCreate( )** – create a named socket-based server

SYNOPSIS

```
SAL_SERVER_ID salCreate
```

```
    (
    const char *              name,          /* service name */
    int                       sockFamily,    /* desired socket address family
*/
    int                       sockType,      /* desired socket type */
    int                       sockProtocol,  /* desired socket protocol */
    const struct salSockopt * options,       /* array of socket options */
    int                       numOptions     /* number of socket options */
    )
```

DESCRIPTION

This routine creates a socket-based server. One or more sockets are created for the server, and the service is registered with SNS using the service name *name*.

name is represented in the following URL format:

[SNS:]service_name[@scope]

Refer to **snsLib** for more information on the format.

This routine tries to create one or more sockets for the combination defined by *sockFamily*, *sockType*, and *sockProtocol*. If the *sockFamily* specified is **AF_UNSPEC**, then a socket creation attempt is made with each family type supported by SAL. If the *sockType* specified is 0, then a socket creation attempt is made with each socket type. If the *sockProtocol* specified is 0, then the default protocol for that family is used.

The *sockFamily*, *sockType*, and *sockProtocol* parameters can be used to limit the server to a given address family and/or socket type and/or socket protocol. salCreate supports connection-oriented message based socket types only, and creates a passive listening socket.

The *options* parameter points to an array of *numOptions* socket option values that are applied to each server socket created. If the socket cannot be successfully configured, it is closed and is not incorporated into the server.

WARNING

Once successfully created, the SAL server must still be configured with one or more processing routines before calling **salRun( )**.

RETURNS

created server ID, **NULL** if fails.

ERRNO

S_salLib_INVALID_ARGUMENT
An invalid argument was passed to this routine.

S_salLib_SERVER_SOCKET_ERROR
Unable to create any sockets with the desired properties

S_salLib_SNS_UNAVAILABLE
Unable to establish connection to the SNS server task.

S_salLib_SNS_DID_NOT_REPLY
Did not receive a reply from the SNS server task.

S_salLib_SNS_PROTOCOL_ERROR
Received an invalid reply from the SNS server task.

S_salLib_SNS_OUT_OF_MEMORY
The SNS server task has insufficient memory to register the service.

S_salLib_SERVICE_ALREADY_EXISTS
The specified service has already been registered with SNS.

SEE ALSO **salServer**, **salDelete( )**, **salRemove( )**, **salServerRtnSet( )**

salDelete( )

NAME **salDelete( )** – delete a named socket-based server

SYNOPSIS

```
STATUS salDelete
    (
    SAL_SERVER_ID server  /* server structure to use */
    )
```

DESCRIPTION This routine deletes the socket-based server specified by *server*. and frees the server data structure memory. All the sockets associated with *server* are closed. The associated service is deregistered from SNS.

A server can only be deleted by the task in the same RTP (or kernel) as the service owner.

RETURNS **OK** or **ERROR**.

ERRNO

S_salLib_INVALID_ARGUMENT
An invalid argument was passed to this routine.

S_salLib_SNS_UNAVAILABLE
Unable to establish connection to the SNS server task.

S_salLib_SNS_DID_NOT_REPLY
Did not receive a reply from the SNS server task.

S_salLib_SNS_PROTOCOL_ERROR
Received an invalid reply from the SNS server task.

S_salLib_INVALID_SERVICE_DESCRIPTOR
Service descriptor is not registered with SNS, or has a different owner.

SEE ALSO **salServer**, **salCreate( )**, **salRemove( )**

salNameFind( )

NAME

salNameFind( ) – find services with the specified name

SYNOPSIS

```
int salNameFind
    (
    const char * pattern,                   /* services name pattern */
    char         servName[][SAL_SERV_NAME_MAXSIZE],
                                            /* buffer to hold the returned
name */
    int          num,                       /* number of element in the
servNames */
    void     **  ppCookie                   /* cookie get/return last
matching address */
    )
```

DESCRIPTION

This function returns services with names that match the specified *pattern*.

Applications provide the buffer for storing the returned names. The function returns the number of names found. The function also returns a cookie for follow up searching.

pattern is represented in the following URL format:

[SNS:]service_name[@scope]

If *pattern* contains wildcard characters, the routine will search for all services that match the pattern.

Refer to **snsLib** for more information on the format and the use of wildcards.

The function returns a number of services no greater than *num*. If more matches are found the function can be called again to retrieve the remaining values. The behavior of the function is determined by the *ppCookie* field.

In order to guarantee all data can be retrieved (possibly through subsequent calls) when the function is called for the first time, the *ppCookie* field needs to be non-**NULL** and the value **ppCookie* needs to be set to **NULL**. If the returned value **ppCookie* is still **NULL**, this means all the services matching the *pattern* have been retrieved. XXX - Yiming to verify If the returned value **ppCookie* is not **NULL**, this means that more matches might be available. In this case, the client application can call **salNameFind( )** again using the returned *ppCookie* to retrieve further entries.

Hence, in order to start a new search, either *ppCookie* is **NULL** (in which case the function can not be called again to retrieve more values) or **ppCookie* is **NULL**.

RETURNS

>=0: number of services found, -1: error.

ERRNO

S_salLib_INVALID_ARGUMENT
Invalid argument.

S_salLib_SNS_UNAVAILABLE
Unable to establish communications with the SNS server task.

SEE ALSO **salClient**, **salSocketFind( )**, **snsLib**

salOpen( )

NAME **salOpen( )** – establish communication with a named socket-based server

SYNOPSIS
```
int salOpen
    (
    const char * name  /* service name in URL format */
    )
```

DESCRIPTION This routine establishes a connection to the server application corresponding to the SNS service name *name*. If the specified service exists **salOpen( )** tries to connect to each of the server's sockets in turn, until it is successful or all sockets have been tried; it returns the resulting socket descriptor.

name is represented in the following URL format:

[SNS:]service_name[@scope]

If *name* contains wildcard characters, the routine will use the first matching service.

Refer to **snsLib** for more information on the format and the use of wildcards.

This routine uses the default socket options for the client socket it creates; if special options are required by the client before completing the connection, use **salSocketFind( )** to establish communication with the server.

User should close the returned socket using **close( )**.

RETURNS >=0: the descriptor of the newly connected socket; -1 : cannot establish communication.

ERRNO
S_salLib_INVALID_ARGUMENT
An invalid argument was passed to this routine.

S_salLib_SNS_UNAVAILABLE
Unable to establish connection to the SNS server task.

S_salLib_SNS_DID_NOT_REPLY
Did not receive a reply from the SNS server task.

S_salLib_SNS_PROTOCOL_ERROR
Received an invalid reply from the SNS server task.

S_salLib_SERVICE_NOT_FOUND
The specified service is not registered with SNS.

S_salLib_INVALID_SERVICE_DESCRIPTOR
The specified service was deregistered from SNS before all socket addresses could be examined.

S_salLib_CLIENT_SOCKET_ERROR
Unable to connect to any of the specified server socket addresses.

SEE ALSO **salClient**, **close( )**, **salSocketFind( )**, **snsLib**

salRemove( )

NAME **salRemove( )** – Remove service from SNS by name

SYNOPSIS

```
STATUS salRemove
    (
    const char * name  /* service name */
    )
```

DESCRIPTION This function removes a service identified by *name* from SNS. Unlike **salDelete( )**, which requires the caller and service owner to be in the same memory space, this function can delete any service as long as the service is visible to the caller. Therefore, a service with scope **node** can be deleted by any task on the same node, and a service with scope **private** can only be deleted by tasks in the same memory space. Further, services of scope **cluster** (or larger) can only be deleted by the node that created them.

name is represented in the following URL format:

[SNS:]service_name[@scope]

Refer to **snsLib** for more information on the format.

name must uniquely identify a service:

service_name
should not contain any wildcard character

scope
must refer a specific level (i.e. the "upto_" prefix can not be used)

NOTE This routine removes only the service name from SNS. It does not remove the service, nor does it close any of the sockets associated to it. These features are provided by **salDelete( )**.

RETURNS **OK** if the service is removed, **ERROR** otherwise.

ERRNO

S_salLib_INVALID_ARGUMENT
The service name is invalid

S_salLib_SERVICE_NOT_FOUND
The specified service is not found.

SEE ALSO

salServer, **salDelete( )**, **salCreate( )**, **snsLib**

salRun( )

NAME

salRun( ) – activate a socket-based server

SYNOPSIS

```
STATUS salRun
    (
    SAL_SERVER_ID server,  /* server structure to use */
    void  *       pData    /* user private data */
    )
```

DESCRIPTION

This routine activates the SAL server specified by *server*. The server monitors all sockets associated with the server, and calls an appropriate processing routine whenever a socket requires attention.

Once invoked, this routine will execute indefinitely and will return only when the server terminates.

Server termination occurs automatically if **salRun( )** detects an error.

The server can terminate also by the application through the processing routine return value **SAL_RUN_TERMINATE**. In this case **salRun( )** simply returns **OK**.

In both cases **salRun( )** does not close any socket. **salDelete( )** should be called to perform the cleanup.

The parameter *pData* can be used to pass any user data. This data is passed to the processing routines when they are being called.

Processing routines should be configured in the server before this routine is called.

RETURNS

OK if server is terminated by processing routine, **ERROR** otherwise.

ERRNO

S_salLib_INVALID_ARGUMENT
An invalid argument was passed to this routine.

S_salLib_SERVER_SOCKET_ERROR
A server socket has failed unexpectedly.

S_salLib_INTERNAL_ERROR
The server's internal data structure has become corrupted.

SEE ALSO **salServer**, **salServerRtnSet( )**

salServerRtnSet( )

NAME **salServerRtnSet( )** – configures the processing routine with the SAL server

SYNOPSIS

```
STATUS salServerRtnSet
    (
    SAL_SERVER_ID svrId,    /* server ID */
    SAL_RTN_TYPE  rtnType,  /* type of processing routine to set */
    SAL_SERV_RTN  routine   /* processing routine entry point */
    )
```

DESCRIPTION This routine configures a processing routine with the server *pSrvrId*. The processing routine is identified by the type *rtnType* and the **SAL_SERV_RTN** function pointer *routine*.

It accepts the following *rtnType*:

SAL_RTN_READ
read routine

SAL_RTN_ACCEPT
accept routine

If *routine* is **NULL**, the processing routine is cleared and the default handler will be used, if available.

This function must be called before activating the SAL server, i.e. before the call to **salRun( )**.

RETURNS **OK** or **ERROR**

ERRNO **S_salLib_INVALID_ARGUMENT**
An invalid argument was passed to this routine.

SEE ALSO **salServer**, **salRun( )**

salSocketFind( )

NAME **salSocketFind( )** – find sockets for a named socket-based server

SYNOPSIS

```
STATUS salSocketFind
    (
    const char *       name,           /* service name in URL format */
    int                 sockFamily,     /* desired socket address family */
    int                 sockType,       /* desired socket type */
    int                 sockProtocol,   /* desired socket protocol */
    struct addrinfo ** ppSockInfoList  /* list of socket entries */
    )
```

DESCRIPTION

This routine looks for sockets related to a server application registered with SNS, which matches the specified search criteria. Each socket entry associated with the SNS service name *name* is examined to see if it is compatible with the restrictions imposed by *sockFamily*, *sockType*, and *sockProtocol*. The search succeeds if at least one matching socket entry is found.

name is represented in the following URL format:

[SNS:]service_name[@scope]

Please refer to **snsLib** for more information on the format.

If *name* contains wildcard characters, the function will only find the first matching service and retrieve its socket information.

To obtain the complete list of service matching the given pattern, use the **salNameFind( )** routine.

If *sockInfoList* is not **NULL** then a list of the matching socket entries is created, and *sockInfoList* is set to the start of the list. However if *sockInfoList* is **NULL**, or the service specified by *name* does not exist, then no list of socket entries is created and *sockInfoList* is left unchanged.

WARNING

The storage for the socket list created by this routine must be released by calling **snsfreeaddrinfo( )** when the list is no longer required.

RETURNS

OK or **ERROR**

ERRNO

S_salLib_INVALID_ARGUMENT
An invalid argument was passed to this routine.

S_salLib_SNS_UNAVAILABLE
Unable to establish connection to the SNS server task.

S_salLib_SNS_DID_NOT_REPLY
Did not receive a reply from the SNS server task.

S_salLib_SNS_PROTOCOL_ERROR
Received an invalid reply from the SNS server task.

S_salLib_SERVICE_NOT_FOUND
The specified service is not registered with SNS.

S_salLib_INVALID_SERVICE_DESCRIPTOR
The specified service was deregistered from SNS before all socket entries could be examined.

S_salLib_NO_SOCKET_FOUND
The specified service has no sockets that match the desired criteria.

SEE ALSO **salClient**, **salNameFind( )**, **snsLib**

sbeRegister( )

NAME **sbeRegister( )** – register with the VxBus subsystem

SYNOPSIS
```
void sbeRegister(void)
```

DESCRIPTION This routine registers the Broadcom driver with VxBus as a child of the PCI bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **sbeVxbEnd**

scMemValEnable( )

NAME **scMemValEnable( )** – enable or disable pointer/buffer validation in system calls

SYNOPSIS
```
void scMemValEnable
    (
    BOOL enable  /* TRUE: enable validation; FALSE: disable validation */
    )
```

DESCRIPTION This routine either enables or disables pointer/buffer validations in system calls, system wide. By default when the OS starts, pointer/buffer validation is ON. Pointer validation can also be disabled for a given RTP by passing the option **RTP_BUFFER_VAL_OFF** as part of the *options* parameter of **rtpSpawn( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scMemVal**, **scMemValidate( )**, **rtpSpawn( )**

scMemValidate( )

NAME

scMemValidate( ) – validate an address range passed to a system call routine

SYNOPSIS

```
STATUS scMemValidate
    (
    const void * addr,   /* start address                        */
    UINT         size,   /* address range size in bytes          */
    SC_PROT_ATTR access  /* minimal access in supervisor mode    */
    )
```

DESCRIPTION

The routine **scMemValidate( )** should be used by system call validation code to verify that a pointer passed as a parameter to a system call routine points to a memory location that belongs to the calling RTP memory context, and that this memory can be dereferenced by the kernel while it executes the system call.

The routine checks if the memory range [*addr* ... (*addr* + size -1)] belongs entirely to either:
- the calling task's stack
- the memory section corresponding to the read-only segment of the RTP: text + rodata segment.
- the memory section corresponding to the data segment of the RTP (read-write access).
- the memory section corresponding to the bss segment of the RTP (read-write access).
- a block that was mapped by the calling RTP vith **mmap( )**. Note that in addition to memory block obtained by direct call to **mmap( )**, this also includes blocks that correspond the RTP heap and shared library private data (data and bss segments) that the RTP is attached to.
- a memory section corresponding to a Shared Data region that the RTP has mapped. Note that this also takes care of the read-only code of SLs that the RTP is attached to.
- other RTP-private memory, such as stack of other tasks in the same RTP.

Note that if *size* is equal to 0, **scMemVal( )** simply returns **OK** without validating the zero-length buffer.

Once the buffer to validate is matched with one of the memory section listed above, that is the buffer is contained entirely within this memory section, **scMemValidate( )** checks that the access permissions to this memory section authorizes the access defined by *access*. The type of access requested is defined by *access* parameter and can take the following values:
- **SC_PROT_READ** (0x1)
- **SC_PROT_WRITE** (0x2)

If *access* is set to 0, then it defaults to (**SC_PROT_READ** | **SC_PROT_WRITE**).

The parameters *addr* and *size* do not need to be aligned to a MMU page size.

Buffer validation accross system calls can be disabled system wide by calling scMemValEnable (**FALSE**). To re-enable buffer validation system-wide simply call scMemValEnable(**TRUE**). Buffer validation accross system calls can be disabled for a given RTP, by passing the option **RTP_BUFFER_VAL_OFF** as part of the *options* parameter when calling **rtpSpawn( )**. For more details refer to the **rtpSpawn( )** manual entry.

RETURNS **OK** if the address range is valid, **ERROR** otherwise.

ERRNO Possible errno generated by this routine include:

EINVAL
The access parameter passed is invalid.

ENOTSUP
The routine was called in unsupported context.

ENOMEM
Memory validation failed with due to boundary constraints.

EACCES
Memory validation failed due to access constraints.

SEE ALSO **scMemVal**, **scMemValEnable( )**, **rtpSpawn( )**

sched_get_priority_max( )

NAME **sched_get_priority_max( )** – get the maximum priority (POSIX)

SYNOPSIS

```
int sched_get_priority_max
    (
    int policy  /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the highest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Maximum priority value, or -1 (**ERROR**) on error.

ERRNO
EINVAL – invalid scheduling policy.

SEE ALSO
schedPxLib

sched_get_priority_min( )

NAME
sched_get_priority_min( ) – get the minimum priority (POSIX)

SYNOPSIS
```
int sched_get_priority_min
    (
    int policy  /* scheduling policy */
    )
```

DESCRIPTION
This routine returns the value of the lowest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE
If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS
Minimum priority value, or -1 (**ERROR**) on error.

ERRNO
EINVAL – invalid scheduling policy.

SEE ALSO
schedPxLib

sched_getparam( )

NAME
sched_getparam( ) – get the scheduling parameters for a specified task (POSIX)

SYNOPSIS
```
int sched_getparam
    (
    pid_t                tid,   /* task ID */
    struct sched_param * param  /* scheduling param to store priority */
    )
```

DESCRIPTION
This routine gets the scheduling priority for a specified task, *tid*. If *tid* is 0, it gets the priority of the calling task. The task's priority is copied to the **sched_param** structure pointed to by *param*.

NOTE

If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

0 (**OK**) if successful, or -1 (**ERROR**) on error.

ERRNO

ESRCH – invalid task ID.

SEE ALSO

schedPxLib

sched_getscheduler( )

NAME

sched_getscheduler( ) – get the current scheduling policy (POSIX)

SYNOPSIS

```
int sched_getscheduler
    (
    pid_t tid  /* task ID */
    )
```

DESCRIPTION

This routine returns the currents scheduling policy (i.e., **SCHED_FIFO** or **SCHED_RR**).

RETURNS

Current scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (**ERROR**) on error.

ERRNO

ESRCH – invalid task ID.

SEE ALSO

schedPxLib

sched_rr_get_interval( )

NAME

sched_rr_get_interval( ) – get the current time slice (POSIX)

SYNOPSIS

```
int sched_rr_get_interval
    (
    pid_t             tid,      /* task ID */
    struct timespec * interval  /* struct to store time slice */
    )
```

DESCRIPTION

This routine sets *interval* to the current time slice period if round-robin scheduling is currently enabled.

RETURNS 0 (**OK**) if successful, -1 (**ERROR**) on error.

ERRNO **EINVAL** – round-robin scheduling is not currently enabled.
ESRCH – invalid task ID.

SEE ALSO **schedPxLib**

sched_setparam( )

NAME **sched_setparam( )** – set a task's priority (POSIX)

SYNOPSIS

```
int sched_setparam
    (
    pid_t                       tid,   /* task ID */
    const struct sched_param * param  /* scheduling parameter */
    )
```

DESCRIPTION This routine sets the priority of a specified task, *tid*. If *tid* is 0, it sets the priority of the calling task. Valid priority numbers are 0 through 255.

The *param* argument is a structure whose member **sched_priority** is the integer priority value. For example, the following program fragment sets the calling task's priority to 13 using POSIX interfaces:

```
#include "sched.h"
 ...
struct sched_param AppSchedPrio;
 ...
AppSchedPrio.sched_priority = 13;
if ( sched_setparam (0, &AppSchedPrio) != OK )
    {
    ... /* recovery attempt or abort message */
    }
 ...
```

NOTE If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS 0 (**OK**) if successful, or -1 (**ERROR**) on error.

ERRNO **EINVAL** – scheduling priority is outside valid range.
ESRCH – task ID is invalid.

SEE ALSO **schedPxLib**

sched_setscheduler( )

NAME

sched_setscheduler( ) – set scheduling policy and scheduling parameters (POSIX)

SYNOPSIS

```
int sched_setscheduler
    (
    pid_t                     tid,     /* task ID */
    int                       policy,  /* scheduling policy requested */
    const struct sched_param * param    /* scheduling parameters requested */
    )
```

DESCRIPTION

This routine sets the scheduling policy and scheduling parameters for a specified task, *tid*. If *tid* is 0, it sets the scheduling policy and scheduling parameters for the calling task.

Because VxWorks does not set scheduling policies (e.g., round-robin scheduling) on a task-by-task basis, setting a scheduling policy that conflicts with the current system policy simply fails and errno is set to **EINVAL**. If the requested scheduling policy is the same as the current system policy, then this routine acts just like **sched_setparam( )**.

NOTE

If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

The previous scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (**ERROR**) on error.

ERRNO

EINVAL – scheduling priority is outside valid range, or it is impossible to set the specified scheduling policy.
ESRCH – invalid task ID.

SEE ALSO

schedPxLib

sched_yield( )

NAME

sched_yield( ) – relinquish the CPU (POSIX)

SYNOPSIS

```
int sched_yield (void)
```

DESCRIPTION

This routine forces the running task to give up the CPU.

RETURNS

0 (**OK**) if successful, or -1 (**ERROR**) on error.

ERRNO Not Available

SEE ALSO **schedPxLib**

scsi2IfInit( )

NAME **scsi2IfInit( )** – initialize the SCSI-2 interface to **scsiLib**

SYNOPSIS

```
void scsi2IfInit (void)
```

DESCRIPTION This routine initializes the SCSI-2 function interface by adding all the routines in **scsi2Lib** plus those in **scsiDirectLib** and **scsiCommonLib**. It is invoked at startup if the component **INCLUDE_SCSI2** is configured in VxWorks. The calling interface remains the same between SCSI-1 and SCSI-2; this routine simply sets the calling interface function pointers to the SCSI-2 functions.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiAutoConfig( )

NAME **scsiAutoConfig( )** – configure all devices connected to a SCSI controller

SYNOPSIS

```
STATUS scsiAutoConfig
    (
    SCSI_CTRL *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine cycles through all valid SCSI bus IDs and logical unit numbers (LUNs), attempting a **scsiPhysDevCreate( )** with default parameters on each. All devices which support the INQUIRY command are configured. The **scsiShow( )** routine can be used to find the system table of SCSI physical devices attached to a specified SCSI controller. In addition, **scsiPhysDevIdGet( )** can be used programmatically to get a pointer to the **SCSI_PHYS_DEV** structure associated with the device at a specified SCSI bus ID and LUN.

RETURNS **OK**, or **ERROR** if *pScsiCtrl* and the global variable **pSysScsiCtrl** are both **NULL**.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiBlkDevCreate( )

NAME **scsiBlkDevCreate( )** – define a logical partition on a SCSI block device

SYNOPSIS

```
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 */
    )
```

DESCRIPTION This routine creates and initializes a **BLK_DEV** structure, which describes a logical partition on a SCSI physical-block device. A logical partition is an array of contiguously addressed blocks; it can be completely described by the number of blocks and the address of the first block in the partition. In normal configurations partitions do not overlap, although such a condition is not an error.

NOTE If *numBlocks* is 0, the rest of device is used.

RETURNS A pointer to the created **BLK_DEV**, or **NULL** if parameters exceed physical device boundaries, if the physical device is not a block device, or if memory is insufficient for the structures.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiBlkDevInit( )

NAME **scsiBlkDevInit( )** – initialize fields in a SCSI logical partition

SYNOPSIS

```
void scsiBlkDevInit
    (
    SCSI_BLK_DEV * pScsiBlkDev,   /* ptr to SCSI block dev. struct */
    int            blksPerTrack,  /* blocks per track */
    int            nHeads         /* number of heads */
    )
```

DESCRIPTION
This routine specifies the disk-geometry parameters required by certain file systems (for example, dosFs). It is called after a **SCSI_BLK_DEV** structure is created with **scsiBlkDevCreate( )**, but before calling a file system initialization routine. It is generally required only for removable-media devices.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
scsiLib

scsiBlkDevShow( )

NAME
scsiBlkDevShow( ) – show the **BLK_DEV** structures on a specified physical device

SYNOPSIS
```
void scsiBlkDevShow
    (
    SCSI_PHYS_DEV * pScsiPhysDev  /* ptr to SCSI physical device info */
    )
```

DESCRIPTION
This routine displays all of the **BLK_DEV** structures created on a specified physical device. This routine is called by **scsiShow( )** but may also be invoked directly, usually from the shell.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
scsiLib, **scsiShow( )**

scsiBusReset( )

NAME
scsiBusReset( ) – pulse the reset signal on the SCSI bus

SYNOPSIS
```
STATUS scsiBusReset
    (
    SCSI_CTRL * pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine calls a controller-specific routine to reset a specified controller's SCSI bus. If no controller is specified (*pScsiCtrl* is 0), the value in the global variable **pSysScsiCtrl** is used.

RETURNS **OK**, or **ERROR** if there is no controller or controller-specific routine.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiCacheSnoopDisable( )

NAME **scsiCacheSnoopDisable( )** – inform SCSI that hardware snooping of caches is disabled

SYNOPSIS

```
void scsiCacheSnoopDisable
    (
    SCSI_CTRL * pScsiCtrl  /* pointer to a SCSI_CTRL structure */
    )
```

DESCRIPTION This routine informs the SCSI library that hardware snooping is disabled and that **scsi2Lib** should execute any neccessary cache coherency code. In order to make **scsi2Lib** aware that hardware snooping is disabled, this routine should be called after all SCSI-2 initializations, especially after **scsi2CtrlInit( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiCacheSnoopEnable( )

NAME **scsiCacheSnoopEnable( )** – inform SCSI that hardware snooping of caches is enabled

SYNOPSIS

```
void scsiCacheSnoopEnable
    (
    SCSI_CTRL * pScsiCtrl  /* pointer to a SCSI_CTRL structure */
    )
```

DESCRIPTION This routine informs the SCSI library that hardware snooping is enabled and that **scsi2Lib** need not execute any cache coherency code. In order to make **scsi2Lib** aware that hardware snooping is enabled, this routine should be called after all SCSI-2 initializations, especially after **scsi2CtrlInit( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiCacheSynchronize( )

NAME **scsiCacheSynchronize( )** – synchronize the caches for data coherency

SYNOPSIS

```
void scsiCacheSynchronize
    (
    SCSI_THREAD *      pThread,  /* ptr to thread info    */
    SCSI_CACHE_ACTION action     /* cache action required */
    )
```

DESCRIPTION This routine performs whatever cache action is necessary to ensure cache coherency with respect to the various buffers involved in a SCSI command.

The process is as follows:

1. The buffers for command, identification, and write data, which are simply written to SCSI, are flushed before the command.
2. The status buffer, which is written and then read, is cleared (flushed and invalidated) before the command.
3. The data buffer for a read command, which is only read, is cleared before the command.

The data buffer for a read command is cleared before the command rather than invalidated after it because it may share dirty cache lines with data outside the read buffer. DMA drivers for older versions of the SCSI library have flushed the first and last bytes of the data buffer before the command. However, this approach is not sufficient with the enhanced SCSI library because the amount of data transferred into the buffer may not fill it, which would cause dirty cache lines which contain correct data for the un-filled part of the buffer to be lost when the buffer is invalidated after the command.

To optimize the performance of the driver in supporting different caching policies, the routine uses the **CACHE_USER_FLUSH** macro when flushing the cache. In the absence of a **CACHE_USER_CLEAR** macro, the following steps are taken:

1. If there is a non-**NULL** flush routine in the **cacheUserFuncs** structure, the cache is cleared.

2. If there is a non-**NULL** invalidate routine, the cache is invalidated.
3. Otherwise nothing is done; the cache is assumed to be coherent without any software intervention.

Finally, since flushing (clearing) cache line entries for a large data buffer can be time-consuming, if the data buffer is larger than a preset (run-time configurable) size, the entire cache is flushed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiErase( )

NAME **scsiErase( )** – issue an ERASE command to a SCSI device

SYNOPSIS

```
STATUS scsiErase
    (
    SCSI_PHYS_DEV *pScsiPhysDev,  /* ptr to SCSI physical device */
    BOOL          longErase       /* TRUE for entire tape erase  */
    )
```

DESCRIPTION This routine issues an ERASE command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiFormatUnit( )

NAME **scsiFormatUnit( )** – issue a **FORMAT_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiFormatUnit
    (
    SCSI_PHYS_DEV * pScsiPhysDev,   /* ptr to SCSI physical device */
    BOOL            cmpDefectList,  /* whether defect list is complete */
    int             defListFormat,  /* defect list format */
    int             vendorUnique,   /* vendor unique byte */
    int             interleave,     /* interleave factor */
    char *          buffer,         /* ptr to input data buffer */
    int             bufLength       /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **FORMAT_UNIT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiIdentMsgBuild( )

NAME **scsiIdentMsgBuild( )** – build an identification message

SYNOPSIS

```
int scsiIdentMsgBuild
    (
    UINT8 *         msg,
    SCSI_PHYS_DEV * pScsiPhysDev,
    SCSI_TAG_TYPE   tagType,
    UINT            tagNumber
    )
```

DESCRIPTION This routine builds an identification message in the caller's buffer, based on the specified physical device, tag type, and tag number.

If the target device does not support messages, there is no identification message to build.

Otherwise, the identification message consists of an IDENTIFY byte plus an optional QUEUE TAG message (two bytes), depending on the type of tag used.

NOTE This function is not intended for use by application programs.

RETURNS The length of the resulting identification message in bytes or -1 for **ERROR**.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiIdentMsgParse( )

NAME **scsiIdentMsgParse( )** – parse an identification message

SYNOPSIS

```
SCSI_IDENT_STATUS scsiIdentMsgParse
    (
    SCSI_CTRL      * pScsiCtrl,
    UINT8          * msg,
    int              msgLength,
    SCSI_PHYS_DEV ** ppScsiPhysDev,
    SCSI_TAG       * pTagNum
    )
```

DESCRIPTION This routine scans a (possibly incomplete) identification message, validating it in the process. If there is an IDENTIFY message, it identifies the corresponding physical device.

If the physical device is currently processing an untagged (ITL) nexus, identification is complete. Otherwise, the identification is complete only if there is a complete QUEUE TAG message.

If there is no physical device corresponding to the IDENTIFY message, or if the device is processing tagged (ITLQ) nexuses and the tag does not correspond to an active thread (it may have been aborted by a timeout, for example), then the identification sequence fails.

The caller's buffers for physical device and tag number (the results of the identification process) are always updated. This is required by the thread event handler (see **scsiMgrThreadEvent( )**.)

NOTE This function is not intended for use by application programs.

RETURNS The identification status (incomplete, complete, or rejected).

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiInquiry( )

NAME **scsiInquiry( )** – issue an INQUIRY command to a SCSI device

SYNOPSIS

```
STATUS scsiInquiry
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device */
    char *          buffer,        /* ptr to input data buffer */
    int              bufLength      /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues an INQUIRY command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiIoctl( )

NAME **scsiIoctl( )** – perform a device-specific I/O control function

SYNOPSIS

```
STATUS scsiIoctl
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI block device info */
    int              function,      /* function code */
    int              arg            /* argument to pass called function */
    )
```

DESCRIPTION This routine performs a specified **ioctl** function using a specified SCSI block device.

RETURNS The status of the request, or **ERROR** if the request is unsupported.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiLoadUnit( )

NAME **scsiLoadUnit( )** – issue a LOAD/UNLOAD command to a SCSI device

SYNOPSIS

```
STATUS scsiLoadUnit
    (
    SCSI_SEQ_DEV * pScsiSeqDev,  /* ptr to SCSI physical device */
    BOOL           load,         /* TRUE=load, FALSE=unload     */
    BOOL           reten,        /* TRUE=retention and unload   */
    BOOL           eot           /* TRUE=end of tape and unload */
    )
```

DESCRIPTION This routine issues a LOAD/UNLOAD command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiMgrBusReset( )

NAME **scsiMgrBusReset( )** – handle a controller-bus reset event

SYNOPSIS

```
void scsiMgrBusReset
    (
    SCSI_CTRL * pScsiCtrl  /* SCSI ctrlr on which bus reset */
    )
```

DESCRIPTION This routine resets in turn: each attached physical device, each target, and the controller-finite-state machine. In practice, this routine implements the SCSI hard reset option.

NOTE This routine does not physically reset the SCSI bus; see **scsiBusReset( )**. This routine should not be called by application programs.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsiMgrLib**

scsiMgrCtrlEvent( )

NAME **scsiMgrCtrlEvent( )** – send an event to the SCSI controller state machine

SYNOPSIS

```
void scsiMgrCtrlEvent
    (
    SCSI_CTRL *     pScsiCtrl,
    SCSI_EVENT_TYPE eventType
    )
```

DESCRIPTION This routine is called by the thread driver whenever selection, reselection, or disconnection occurs or when a thread is activated. It manages a simple finite-state machine for the SCSI controller.

NOTE This function should not be called by application programs.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsiMgrLib**

scsiMgrEventNotify( )

NAME **scsiMgrEventNotify( )** – notify the SCSI manager of a SCSI (controller) event

SYNOPSIS

```
STATUS scsiMgrEventNotify
    (
    SCSI_CTRL *  pScsiCtrl,  /* pointer to SCSI controller structure */
    SCSI_EVENT * pEvent,     /* pointer to the SCSI event */
    int          eventSize   /* size of the event information */
    )
```

DESCRIPTION This routine posts an event message on the appropriate SCSI manager queue, then notifies the SCSI manager that there is a message to be accepted.

NOTE This routine should not be called by application programs.

No access serialization is required, because event messages are only posted by the SCSI controller ISR. See the reference entry for **scsiBusResetNotify( )**.

RETURNS **OK**, or **ERROR** if the SCSI manager's event queue is full.

ERRNO Not Available

SEE ALSO **scsiMgrLib**, **scsiBusResetNotify( )**

scsiMgrShow( )

NAME

scsiMgrShow( ) – show status information for the SCSI manager

SYNOPSIS

```
void scsiMgrShow
    (
    SCSI_CTRL * pScsiCtrl,       /* SCSI controller to use        */
    BOOL        showPhysDevs,    /* TRUE => show phys dev details */
    BOOL        showThreads,     /* TRUE => show thread details   */
    BOOL        showFreeThreads  /* TRUE => show free thread IDs  */
    )
```

DESCRIPTION

This routine shows the current state of the SCSI manager for the specified controller, including the total number of threads created and the number of threads currently free.

Optionally, this routine also shows details for all created physical devices on this controller and all threads for which SCSI requests are outstanding. It also shows the IDs of all free threads.

NOTE

The information displayed is volatile; this routine is best used when there is no activity on the SCSI bus. Threads allocated by a client but for which there are no outstanding SCSI requests are not shown.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

scsiMgrLib

scsiMgrThreadEvent( )

NAME

scsiMgrThreadEvent( ) – send an event to the thread state machine

SYNOPSIS

```
void scsiMgrThreadEvent
    (
    SCSI_THREAD *           pThread,
    SCSI_THREAD_EVENT_TYPE eventType
    )
```

DESCRIPTION

This routine forwards an event to the thread's physical device. If the event is completion or deferral, it frees up the tag which was allocated when the thread was activated and either completes or defers the thread.

NOTE
This function should not be called by application programs.

The thread passed into this function does not have to be an active client thread (it may be an identification thread).

If the thread has no corresponding physical device, this routine does nothing. (This occasionally occurs if an unexpected disconnection or bus reset happens when an identification thread has not yet identified which physical device it corresponds to.)

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
scsiMgrLib

scsiModeSelect( )

NAME
scsiModeSelect( ) – issue a **MODE_SELECT** command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSelect
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device
*/
    int             pageFormat,    /* value of the page format bit (0-1)
*/
    int             saveParams,    /* value of the save parameters bit (0-1)
*/
    char *          buffer,        /* ptr to output data buffer
*/
    int             bufLength      /* length of buffer in bytes
*/
    )
```

DESCRIPTION
This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS
OK, or **ERROR** if the command fails.

ERRNO
Not Available

SEE ALSO
scsiLib

scsiModeSense( )

NAME
scsiModeSense( ) – issue a **MODE_SENSE** command to a SCSI device

SYNOPSIS
```
STATUS scsiModeSense
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device */
    int             pageControl,   /* value of the page control field (0-3)
*/
    int             pageCode,      /* value of the page code field (0-0x3f)
*/
    char *          buffer,        /* ptr to input data buffer */
    int             bufLength      /* length of buffer in bytes */
    )
```

DESCRIPTION
This routine issues a **MODE_SENSE** command to a specified SCSI device.

RETURNS
OK, or **ERROR** if the command fails.

ERRNO
Not Available

SEE ALSO
scsiLib

scsiMsgInComplete( )

NAME
scsiMsgInComplete( ) – handle a complete SCSI message received from the target

SYNOPSIS
```
STATUS scsiMsgInComplete
    (
    SCSI_CTRL   *pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD *pThread     /* ptr to thread info          */
    )
```

DESCRIPTION
This routine parses the complete message and takes any necessary action, which may include setting up an outgoing message in reply. If the message is not understood, the routine rejects it and returns an **ERROR** status.

NOTE
This function is intended for use only by SCSI controller drivers.

RETURNS
OK, or **ERROR** if the message is not supported.

ERRNO
Not Available

SEE ALSO **scsi2Lib**

scsiMsgOutComplete( )

NAME **scsiMsgOutComplete( )** – perform post-processing after a SCSI message is sent

SYNOPSIS

```
STATUS scsiMsgOutComplete
    (
    SCSI_CTRL   *pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD *pThread     /* ptr to thread info          */
    )
```

DESCRIPTION This routine parses the complete message and takes any necessary action.

NOTE This function is intended for use only by SCSI controller drivers.

RETURNS **OK**, or **ERROR** if the message is not supported.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiMsgOutReject( )

NAME **scsiMsgOutReject( )** – perform post-processing when an outgoing message is rejected

SYNOPSIS

```
void scsiMsgOutReject
    (
    SCSI_CTRL   *pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD *pThread     /* ptr to thread info          */
    )
```

NOTE This function is intended for use only by SCSI controller drivers.

RETURNS **OK**, or **ERROR** if the message is not supported.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiPhysDevCreate( )

NAME

scsiPhysDevCreate( ) – create a SCSI physical device structure

SYNOPSIS

```
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 */
    )
```

DESCRIPTION

This routine enables access to a SCSI device and must be the first routine invoked. It must be called once for each physical device on the SCSI bus.

If *reqSenseLength* is **NULL** (0), one or more **REQUEST_SENSE** commands are issued to the device to determine the number of bytes of sense data it typically returns. Note that if the device returns variable amounts of sense data depending on its state, you must consult the device manual to determine the maximum amount of sense data that can be returned.

If *devType* is NONE (-1), an INQUIRY command is issued to determine the device type; as an added benefit, it acquires the device's make and model number. The **scsiShow( )** routine displays this information. Common values of *devType* can be found in **scsiLib.h** or in the SCSI specification.

If *numBlocks* or *blockSize* are specified as **NULL** (0), a **READ_CAPACITY** command is issued to determine those values. This occurs only for device types that support **READ_CAPACITY**.

RETURNS

A pointer to the created **SCSI_PHYS_DEV** structure, or **NULL** if the routine is unable to create the physical-device structure.

ERRNO

Not Available

SEE ALSO

scsiLib

scsiPhysDevDelete( )

NAME

scsiPhysDevDelete( ) – delete a SCSI physical-device structure

SYNOPSIS

```
STATUS scsiPhysDevDelete
    (
    SCSI_PHYS_DEV *pScsiPhysDev  /* ptr to SCSI physical device info */
    )
```

DESCRIPTION

This routine deletes a specified SCSI physical-device structure.

RETURNS

OK, or **ERROR** if **pScsiPhysDev** is **NULL** or SCSI_BLK_DEVs have been created on the device.

ERRNO

Not Available

SEE ALSO

scsiLib

scsiPhysDevIdGet( )

NAME

scsiPhysDevIdGet( ) – return a pointer to a **SCSI_PHYS_DEV** structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevIdGet
    (
    SCSI_CTRL * pScsiCtrl,  /* ptr to SCSI controller info  */
    int         devBusId,   /* device's SCSI bus ID         */
    int         devLUN      /* device's logical unit number */
    )
```

DESCRIPTION

This routine returns a pointer to the **SCSI_PHYS_DEV** structure of the SCSI physical device located at a specified bus ID (*devBusId*) and logical unit number (*devLUN*) and attached to a specified SCSI controller (*pScsiCtrl*).

RETURNS

A pointer to the specified **SCSI_PHYS_DEV** structure, or **NULL** if the structure does not exist.

ERRNO

Not Available

SEE ALSO

scsiLib

scsiPhysDevShow( )

NAME **scsiPhysDevShow( )** – show status information for a physical device

SYNOPSIS

```
void scsiPhysDevShow
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* physical device to be displayed */
    BOOL            showThreads,   /* show IDs of associated threads  */
    BOOL            noHeader       /* do not print title line         */
    )
```

DESCRIPTION This routine shows the state, the current nexus type, the current tag number, the number of tagged commands in progress, and the number of waiting and active threads for a SCSI physical device. Optionally, it shows the IDs of waiting and active threads, if any. This routine may be called at any time, but note that all of the information displayed is volatile.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiRdSecs( )

NAME **scsiRdSecs( )** – read sector(s) from a SCSI block device

SYNOPSIS

```
STATUS scsiRdSecs
    (
    SCSI_BLK_DEV * pScsiBlkDev,  /* ptr to SCSI block device info */
    int            sector,       /* sector number to be read */
    int            numSecs,      /* total sectors to be read */
    char *         buffer        /* ptr to input data buffer */
    )
```

DESCRIPTION This routine reads the specified physical sector(s) from a specified physical device.

RETURNS **OK**, or **ERROR** if the sector(s) cannot be read.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiRdTape( )

NAME

scsiRdTape( ) – read bytes or blocks from a SCSI tape device

SYNOPSIS

```
int scsiRdTape
    (
    SCSI_SEQ_DEV *pScsiSeqDev,  /* ptr to SCSI sequential device info */
    UINT         count,         /* total bytes or blocks to be read   */
    char         *buffer,       /* ptr to input data buffer           */
    BOOL         fixedSize      /* if variable size blocks            */
    )
```

DESCRIPTION

This routine reads the specified number of bytes or blocks from a specified physical device. If the boolean *fixedSize* is true, then *numBytes* represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = **FALSE**), then *numBytes* represents the actual number of bytes to be read.

RETURNS

Number of bytes or blocks actually read, 0 if **EOF**, or **ERROR**.

ERRNO

Not Available

SEE ALSO

scsiSeqLib

scsiReadCapacity( )

NAME

scsiReadCapacity( ) – issue a **READ_CAPACITY** command to a SCSI device

SYNOPSIS

```
STATUS scsiReadCapacity
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device */
    int *           pLastLBA,      /* where to return last */
                                   /* logical block address */
    int *           pBlkLength     /* where to return block length */
    )
```

DESCRIPTION

This routine issues a **READ_CAPACITY** command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

ERRNO

Not Available

SEE ALSO

scsiLib

scsiRelease( )

NAME **scsiRelease( )** – issue a RELEASE command to a SCSI device

SYNOPSIS

```
STATUS scsiRelease
    (
    SCSI_PHYS_DEV *pScsiPhysDev  /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a RELEASE command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiDirectLib**

scsiReleaseUnit( )

NAME **scsiReleaseUnit( )** – issue a RELEASE UNIT command to a SCSI device

SYNOPSIS

```
STATUS scsiReleaseUnit
    (
    SCSI_SEQ_DEV *pScsiSeqDev  /* ptr to SCSI sequential device */
    )
```

DESCRIPTION This routine issues a RELEASE UNIT command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiReqSense( )

NAME **scsiReqSense( )** – issue a **REQUEST_SENSE** command to a SCSI device and read results

SYNOPSIS

```
STATUS scsiReqSense
```

```
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device */
    char *          buffer,        /* ptr to input data buffer */
    int             bufLength      /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **REQUEST_SENSE** command to a specified SCSI device and reads the results.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiReserve( )

NAME **scsiReserve( )** – issue a RESERVE command to a SCSI device

SYNOPSIS

```
STATUS scsiReserve
    (
    SCSI_PHYS_DEV *pScsiPhysDev  /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a RESERVE command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiDirectLib**

scsiReserveUnit( )

NAME **scsiReserveUnit( )** – issue a RESERVE UNIT command to a SCSI device

SYNOPSIS

```
STATUS scsiReserveUnit
    (
    SCSI_SEQ_DEV *pScsiSeqDev  /* ptr to SCSI sequential device */
    )
```

DESCRIPTION This routine issues a RESERVE UNIT command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiRewind( )

NAME **scsiRewind( )** – issue a REWIND command to a SCSI device

SYNOPSIS

```
STATUS scsiRewind
    (
    SCSI_SEQ_DEV *pScsiSeqDev  /* ptr to SCSI Sequential device */
    )
```

DESCRIPTION This routine issues a REWIND command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiSeqDevCreate( )

NAME **scsiSeqDevCreate( )** – create a SCSI sequential device

SYNOPSIS

```
SEQ_DEV *scsiSeqDevCreate
    (
    SCSI_PHYS_DEV *pScsiPhysDev  /* ptr to SCSI physical device info */
    )
```

DESCRIPTION This routine creates a SCSI sequential device and saves a pointer to this **SEQ_DEV** in the SCSI physical device. The following functions are initialized in this structure:

sd_seqRd	**scsiRdTape( )**
sd_seqWrt	**scsiWrtTape( )**
sd_ioctl	**scsiIoctl( )** (in **scsiLib**)

sd_seqWrtFileMarks	**scsiWrtFileMarks( )**
sd_statusChk	**scsiSeqStatusCheck( )**
sd_reset	(not used)
sd_rewind	**scsiRewind( )**
sd_reserve	**scsiReserve( )**
sd_release	**scsiRelease( )**
sd_readBlkLim	**scsiSeqReadBlockLimits( )**
sd_load	**scsiLoadUnit( )**
sd_space	**scsiSpace( )**
sd_erase	**scsiErase( )**

Only one **SEQ_DEV** per **SCSI_PHYS_DEV** is allowed, unlike BLK_DEVs where an entire list is maintained. Therefore, this routine can be called only once per creation of a sequential device.

RETURNS
A pointer to the **SEQ_DEV** structure, or **NULL** if the command fails.

ERRNO
Not Available

SEE ALSO
scsiSeqLib

scsiSeqIoctl( )

NAME
scsiSeqIoctl( ) – perform an I/O control function for sequential access devices

SYNOPSIS
```
int scsiSeqIoctl
    (
    SCSI_SEQ_DEV * pScsiSeqDev,  /* ptr to SCSI sequential device */
    int            function,     /* ioctl function code */
    int            arg           /* argument to pass to called function */
    )
```

DESCRIPTION
This routine issues **scsiSeqLib** commands to perform sequential device-specific I/O control operations.

RETURNS
OK or **ERROR**.

ERRNO
S_scsiLib_INVALID_BLOCK_SIZE

SEE ALSO
scsiSeqLib

scsiSeqReadBlockLimits( )

NAME

scsiSeqReadBlockLimits( ) – issue a **READ_BLOCK_LIMITS** command to a SCSI device

SYNOPSIS

```
STATUS scsiSeqReadBlockLimits
    (
    SCSI_SEQ_DEV * pScsiSeqDev,       /* ptr to SCSI sequential device
*/
    int            *pMaxBlockLength,  /* where to return maximum block length
*/
    UINT16         *pMinBlockLength   /* where to return minimum block length
*/
    )
```

DESCRIPTION

This routine issues a **READ_BLOCK_LIMITS** command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

ERRNO

Not Available

SEE ALSO

scsiSeqLib

scsiSeqStatusCheck( )

NAME

scsiSeqStatusCheck( ) – detect a change in media

SYNOPSIS

```
STATUS scsiSeqStatusCheck
    (
    SCSI_SEQ_DEV *pScsiSeqDev  /* ptr to a sequential dev */
    )
```

DESCRIPTION

This routine issues a **TEST_UNIT_READY** command to a SCSI device to detect a change in media. It is called by file systems before executing **open( )** or **creat( )**.

RETURNS

OK or **ERROR**.

ERRNO

Not Available

SEE ALSO

scsiSeqLib

scsiShow( )

NAME

scsiShow( ) – list the physical devices attached to a SCSI controller

SYNOPSIS

```
STATUS scsiShow
    (
    SCSI_CTRL *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION

This routine displays the SCSI bus ID, logical unit number (LUN), vendor ID, product ID, firmware revision (rev.), device type, number of blocks, block size in bytes, and a pointer to the associated **SCSI_PHYS_DEV** structure for each physical SCSI device known to be attached to a specified SCSI controller.

NOTE

If *pScsiCtrl* is **NULL**, the value of the global variable **pSysScsiCtrl** is used, unless it is also **NULL**.

RETURNS

OK, or **ERROR** if both *pScsiCtrl* and **pSysScsiCtrl** are **NULL**.

ERRNO

Not Available

SEE ALSO

scsiLib

scsiSpace( )

NAME

scsiSpace( ) – move the tape on a specified physical SCSI device

SYNOPSIS

```
STATUS scsiSpace
    (
    SCSI_SEQ_DEV * pScsiSeqDev,  /* ptr to SCSI sequential device info */
    int            count,        /* count for space command            */
    int            spaceCode     /* code for the type of space command */
    )
```

DESCRIPTION

This routine moves the tape on a specified SCSI physical device. There are two types of space code that are mandatory in SCSI; currently these are the only two supported:

Code	Description	Support
000	Blocks	Yes
001	File marks	Yes
010	Sequential file marks	No
011	End-of-data	No
100	Set marks	No

Code	Description	Support
101	Sequential set marks	No

RETURNS **OK**, or **ERROR** if an error is returned by the device.

ERRNO **S_scsiLib_ILLEGAL_REQUEST**

SEE ALSO **scsiSeqLib**

scsiStartStopUnit( )

NAME **scsiStartStopUnit( )** – issue a **START_STOP_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiStartStopUnit
    (
    SCSI_PHYS_DEV *pScsiPhysDev,  /* ptr to SCSI physical device */
    BOOL          start           /* TRUE == start, FALSE == stop */
    )
```

DESCRIPTION This routine issues a **START_STOP_UNIT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiDirectLib**

scsiSyncXferNegotiate( )

NAME **scsiSyncXferNegotiate( )** – initiate or continue negotiating transfer parameters

SYNOPSIS

```
void scsiSyncXferNegotiate
    (
    SCSI_CTRL            *pScsiCtrl,    /* ptr to SCSI controller info  */
    SCSI_TARGET          *pScsiTarget,  /* ptr to SCSI target info      */
    SCSI_SYNC_XFER_EVENT eventType      /* tells what has just happened */
    )
```

DESCRIPTION This routine manages negotiation by means of a finite-state machine which is driven by "significant events" such as incoming and outgoing messages. Each SCSI target has its own independent state machine.

NOTE If the controller does not support synchronous transfer or if the target's maximum REQ/ACK offset is zero, attempts to initiate a round of negotiation are ignored.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiTapeModeSelect( )

NAME **scsiTapeModeSelect( )** – issue a **MODE_SELECT** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSelect
    (
    SCSI_PHYS_DEV *pScsiPhysDev,  /* ptr to SCSI physical device
*/
    int            pageFormat,     /* value of the page format bit (0-1)
*/
    int            saveParams,     /* value of the save parameters bit (0-1)
*/
    char           *buffer,        /* ptr to output data buffer
*/
    int            bufLength       /* length of buffer in bytes
*/
    )
```

DESCRIPTION This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiTapeModeSense( )

NAME **scsiTapeModeSense( )** – issue a **MODE_SENSE** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSense
    (
    SCSI_PHYS_DEV *pScsiPhysDev,  /* ptr to SCSI physical device           */
    int           pageControl,    /* value of the page control field (0-3) */
    int           pageCode,       /* value of the page code field (0-0x3f) */
    char          *buffer,        /* ptr to input data buffer              */
    int           bufLength       /* length of buffer in bytes             */
    )
```

DESCRIPTION This routine issues a **MODE_SENSE** command to a specified SCSI tape device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiTargetOptionsGet( )

NAME **scsiTargetOptionsGet( )** – get options for one or all SCSI targets

SYNOPSIS

```
STATUS scsiTargetOptionsGet
    (
    SCSI_CTRL    *pScsiCtrl,  /* ptr to SCSI controller info */
    int          devBusId,    /* target to interrogate       */
    SCSI_OPTIONS *pOptions    /* buffer to return options    */
    )
```

DESCRIPTION This routine copies the current options for the specified target into the caller's buffer.

RETURNS **OK**, or **ERROR** if the bus ID is invalid.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiTargetOptionsSet( )

NAME

scsiTargetOptionsSet( ) – set options for one or all SCSI targets

SYNOPSIS

```
STATUS scsiTargetOptionsSet
    (
    SCSI_CTRL    *pScsiCtrl,  /* ptr to SCSI controller info   */
    int          devBusId,    /* target to affect, or all      */
    SCSI_OPTIONS *pOptions,   /* buffer containing new options */
    UINT         which        /* which options to change       */
    )
```

DESCRIPTION

This routine sets the options defined by the bitmask **which** for the specified target (or all targets if **devBusId** is **SCSI_SET_OPT_ALL_TARGETS**).

The bitmask **which** can be any combination of the following, bitwise OR'd together (corresponding fields in the **SCSI_OPTIONS** structure are shown in parentheses):

SCSI_SET_OPT_TIMEOUT	**selTimeOut**	select timeout period, microseconds
SCSI_SET_OPT_MESSAGES	**messages**	**FALSE** to disable SCSI messages
SCSI_SET_OPT_DISCONNECT	**disconnect**	**FALSE** to disable discon/recon
SCSI_SET_OPT_XFER_PARAMS	**maxOffset,**	max sync xfer offset, 0=>async
	minPeriod	min sync xfer period, x 4 nsec.
SCSI_SET_OPT_TAG_PARAMS	**tagType,**	default tag type (**SCSI_TAG_***)
	maxTags	max cmd tags available
SCSI_SET_OPT_WIDE_PARAMS	**xferWidth**	data transfer width setting.
		xferWidth = 0 ; 8 bits wide
		xferWidth = 1 ; 16 bits wide

NOTE

This routine can be used after the target device has already been used; in this case, however, it is not possible to change the tag parameters. This routine must not be used while there is any SCSI activity on the specified target(s).

RETURNS

OK, or **ERROR** if the bus ID or options are invalid.

ERRNO

Not Available

SEE ALSO

scsi2Lib

scsiTargetOptionsShow( )

NAME **scsiTargetOptionsShow( )** – display options for specified SCSI target

SYNOPSIS

```
STATUS scsiTargetOptionsShow
    (
    SCSI_CTRL *pScsiCtrl,  /* ptr to SCSI controller info */
    int       devBusId     /* target to interrogate       */
    )
```

DESCRIPTION This routine displays the current target options for the specified target in the following format:

```
Target Options (id <scsi bus ID>):
    selection TimeOut: <timeout> nano secs
     messages allowed: TRUE or FALSE
   disconnect allowed: TRUE or FALSE
       REQ/ACK offset: <negotiated offset>
      transfer period: <negotiated period>
       transfer width: 8 or 16 bits
maximum transfer rate: <peak transfer rate> MB/sec
             tag type: <tag type>
         maximum tags: <max tags>
```

RETURNS **OK**, or **ERROR** if the bus ID is invalid.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiTestUnitRdy( )

NAME **scsiTestUnitRdy( )** – issue a **TEST_UNIT_READY** command to a SCSI device

SYNOPSIS

```
STATUS scsiTestUnitRdy
    (
    SCSI_PHYS_DEV * pScsiPhysDev  /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a **TEST_UNIT_READY** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiThreadInit( )

NAME **scsiThreadInit( )** – perform generic SCSI thread initialization

SYNOPSIS
```
STATUS scsiThreadInit
    (
    SCSI_THREAD * pThread
    )
```

DESCRIPTION This routine initializes the controller-independent parts of a thread structure, which are specific to the SCSI manager.

NOTE This function should not be called by application programs. It is intended to be used by SCSI controller drivers.

RETURNS **OK**, or **ERROR** if the thread cannot be initialized.

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiWideXferNegotiate( )

NAME **scsiWideXferNegotiate( )** – initiate or continue negotiating wide parameters

SYNOPSIS
```
void scsiWideXferNegotiate
    (
    SCSI_CTRL            *pScsiCtrl,    /* ptr to SCSI controller info  */
    SCSI_TARGET          *pScsiTarget,  /* ptr to SCSI target info      */
    SCSI_WIDE_XFER_EVENT eventType      /* tells what has just happened */
    )
```

DESCRIPTION This routine manages negotiation means of a finite-state machine which is driven by "significant events" such as incoming and outgoing messages. Each SCSI target has its own independent state machine.

NOTE If the controller does not support wide transfers or the target's transfer width is zero, attempts to initiate a round of negotiation are ignored; this is because zero is the default narrow transfer.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

ERRNO Not Available

SEE ALSO **scsi2Lib**

scsiWrtFileMarks( )

NAME **scsiWrtFileMarks( )** – write file marks to a SCSI sequential device

SYNOPSIS

```
STATUS scsiWrtFileMarks
    (
    SCSI_SEQ_DEV * pScsiSeqDev,  /* ptr to SCSI sequential device info */
    int            numMarks,     /* number of file marks to write      */
    BOOL           shortMark     /* TRUE to write short file mark      */
    )
```

DESCRIPTION This routine writes file marks to a specified physical device.

RETURNS **OK**, or **ERROR** if the file mark cannot be written.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

scsiWrtSecs( )

NAME **scsiWrtSecs( )** – write sector(s) to a SCSI block device

SYNOPSIS

```
STATUS scsiWrtSecs
    (
    SCSI_BLK_DEV * pScsiBlkDev,  /* ptr to SCSI block device info */
    int            sector,       /* sector number to be written */
    int            numSecs,      /* total sectors to be written */
    char *          buffer         /* ptr to input data buffer */
    )
```

DESCRIPTION This routine writes the specified physical sector(s) to a specified physical device.

RETURNS **OK**, or **ERROR** if the sector(s) cannot be written.

ERRNO Not Available

SEE ALSO **scsiLib**

scsiWrtTape( )

NAME **scsiWrtTape( )** – write data to a SCSI tape device

SYNOPSIS

```
STATUS scsiWrtTape
    (
    SCSI_SEQ_DEV *pScsiSeqDev,  /* ptr to SCSI sequential device info  */
    int          numBytes,      /* total bytes or blocks to be written */
    char         *buffer,       /* ptr to input data buffer            */
    BOOL         fixedSize      /* if variable size blocks             */
    )
```

DESCRIPTION This routine writes data to the current block on a specified physical device. If the boolean *fixedSize* is true, then *numBytes* represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = **FALSE**), then *numBytes* represents the actual number of bytes to be written. If *numBytes* is greater than the **maxBytesLimit** field defined in the **pScsiPhysDev** structure, then more than one SCSI transaction is used to transfer the data.

RETURNS **OK**, or **ERROR** if the data cannot be written or zero bytes are written.

ERRNO Not Available

SEE ALSO **scsiSeqLib**

sdCreate( )

NAME

sdCreate( ) – create a new shared data region

SYNOPSIS

```
SD_ID sdCreate
    (
    char *   name,         /* name of the shared data region */
    int      options,      /* creation options */
    UINT32   size,         /* size of shared data in bytes */
    off_t64  physAddress,  /* optional physical address */
    MMU_ATTR attr,         /* allowed user MMU attributes */
    void **  pVirtAddress  /* optional virtual base address */
    )
```

DESCRIPTION

This routine creates a new shared data region and maps it into the calling task's memory context. The following table shows each parameter and whether it is required or not:

Parameter	Required?	Default
name	Yes	N/A
options	No	0
size	Yes	N/A
physAddress	No	System Allocated
attr	No	Read/Write, System Default Cache Setting
pVirtAddress	Yes	N/A

Because each shared data region must have a unique name, if the region specified by *name* already exists in the system the creation will fail. **NULL** will be returned.

Currently there are only two possible values of *options*:

Option name	Value	Meaning
SD_LINGER	0x1	SD region may remain after the last client unmaps.
SD_PRIVATE	0x2	SD region is only available in the owner RTP.

The value of *size* must be greater than 0. It is rounded up to a page aligned size determined by the architecture.

If *physAddress* is specified and the address is not available, **NULL** will be returned. The *physAddress* specified must be aligned on the architecture dependent page size boundary and must not be mapped to any other memory context.

The MMU attributes specified in *attr* will be used as the default attributes of the shared data region. All client applications will use these by default, and may only change the local access permissions to a subset of these. The application which creates the region will have read and write access in addition to the defaults and will be allowed to set local permissions to any allowed by the architecture.

Basic MMU attribute definitions for shared data regions are provided in the **sdLibCommon.h** header file. These include:

Attribute	Meaning
SD_ATTR_RW	Read/Write for both Supervisor and User Modes
SD_ATTR_RO	Read Only for both Supervisor and User Modes
SD_ATTR_RWX	Read/Write/Execute for both Supervisor and User Modes
SD_ATTR_RX	Read/Execute for both Supervisor and User Modes
SD_CACHE_COPYBACK	Copyback cache mode
SD_CACHE_WRITETHROUGH	Write through cache mode
SD_CACHE_OFF	Cache Off

One of each the **SD_ATTR** and **SD_CACHE** macros above must be provided. The **SD_CACHE** macros can not be combined.

The cache attributes of a shared data region can not be changed after creation. All clients of that region will use the value provided at create time, including the owner.

If more specific MMU attributes are required please see **vmLibCommon.h** for a complete list of available MMU attributes.

NOTE

The **MMU_ATTR** mask used internally by the shared data library is the combination of:

MMU_ATTR_PROT_MASK

MMU_ATTR_VALID_MSK

MMU_ATTR_SPL_MSK

Care must be taken to provide suitable values for all these attributes.

The start address of the shared data region is stored at the location specified by *pVirtAddress*. This must be a valid address within the context of the calling application. It can not be **NULL**.

The **SD_ID** returned is private to the calling application. It can be shared between tasks within that application but not with tasks that reside outside that application.

RETURNS

ID of new shared data region, or **NULL** on error.

ERRNO

Possible errno values set by this routine are:

S_sdLib_VIRT_ADDR_PTR_IS_NULL
pVirtAddress is **NULL**

S_sdLib_ADDR_NOT_ALIGNED
physAddress is not properly aligned

S_sdLib_PHYS_ADDR_OUT_OF_RANGE
physAddress exceeds physical address space

S_sdLib_SIZE_IS_NULL
size is **NULL**

S_sdLib_INVALID_OPTIONS
options is not a valid combination

S_sdLib_VIRT_PAGES_NOT_AVAILABLE
not enough virtual space left in system

S_sdLib_PHYS_PAGES_NOT_AVAILABLE
not enough physical memory left in system

SEE ALSO **sdLib**, **sdOpen( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

sdCreateHookAdd( )

NAME **sdCreateHookAdd( )** – add a hook routine to be called at Shared Data creation

SYNOPSIS

```
STATUS sdCreateHookAdd
    (
    SD_CREATE_HOOK sdCreateHook,  /* hook routine to call */
    BOOL           addToHead      /* add routine to head of list */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called just after an SD is created. The hook routine should have the following prototype:

```
STATUS sdCreateHook
    (
    const SD_ID  sdId,        /* ID of the created SD */
    )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

Shared Data create hooks are called from **sdCreate( )** or **sdOpen( )** after the creation is done. Create hooks are not expected to return anything (return values if any are not checked).

RETURNS **OK**, or **ERROR** if the table of SD create routines is full.

ERRNO N/A.

SEE ALSO **sdLib**, **sdCreateHookDelete( )**

sdCreateHookDelete( )

NAME **sdCreateHookDelete( )** – delete a Shared Data creation hook routine

SYNOPSIS

```
STATUS sdCreateHookDelete
    (
    SD_CREATE_HOOK sdCreateHook  /* hook routine to delete */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of Shared Data create hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **sdLib**, **sdCreateHookAdd( )**

sdDelete( )

NAME **sdDelete( )** – delete a shared data region

SYNOPSIS

```
STATUS sdDelete
    (
    SD_ID sdId,    /* ID of shared data region to delete */
    int   options  /* options field is not used */
    )
```

DESCRIPTION Deletes a shared data region. This is only possible if there are no applications that have the shared data region mapped. Currently there are no options defined for this function, this parameter should be passed as zero always.

Unless the option **SD_LINGER** was specified at creation of the shared data region it will automatically be deleted when the last client application exits or explicitly calls **sdUnmap( )**.

RETURNS **OK**, or **ERROR** on failure.

ERRNO Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

S_sdLib_CLIENT_COUNT_NOT_NULL
sdId still mapped by an application

SEE ALSO **sdLib**, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, **sdProtect( )**

sdDeleteHookAdd( )

NAME **sdDeleteHookAdd( )** – add a hook routine to be called at Shared Data deletion

SYNOPSIS

```
STATUS sdDeleteHookAdd
    (
    SD_DELETE_HOOK sdDeleteHook,  /* hook routine to call */
    BOOL           addToHead      /* add routine to head of list */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called just before a SD is deleted. The hook routine should have the following prototype:

```
void sdDeleteHook
    (
    const SD_ID  sdId,        /* ID of the deleted SD */
    )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

SD delete hooks are called from **sdDelete( )** before any deletion is done. Delete hooks are not expected to return anything (return values if any are not checked).

RETURNS **OK**, or **ERROR** if the table of SD create routines is full.

ERRNO N/A.

SEE ALSO **sdLib**, **sdDeleteHookDelete( )**

sdDeleteHookDelete( )

NAME **sdDeleteHookDelete( )** – delete a Shared Data deletion hook routine

SYNOPSIS

```
STATUS sdDeleteHookDelete
    (
    SD_DELETE_HOOK sdDeleteHook  /* hook routine to delete */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of Shared Data delete hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **sdLib**, **sdDeleteHookAdd( )**

sdGenericHookAdd( )

NAME **sdGenericHookAdd( )** – add a hook routine to be called before Shared Data routine

SYNOPSIS

```
STATUS sdGenericHookAdd
    (
    SD_GENERIC_HOOK sdGenericHook,  /* hook routine to call */
    BOOL            addToHead       /* add routine to head of list */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called just before an SD is created, mapped, unmapped, deleted, or has its protection attributes changed. The hook routine should have the following prototype:

```
STATUS sdGenericHook
    (
    const void *  sdId or name,  /* ID or name of the SD */
    int options                  /* options passed to hook routine */
    )
```

The *options* argument is used to identify what routine invoked the hook and whether the first argument is to be treated as a name or an ID. These are specified by the following enumeration:

```
typedef enum sd_routines
    {
    SD_HOOK_TYPE_MSK    = 0x00000001,
    SD_HOOK_ID          = 0x00000000,
    SD_HOOK_NAME        = 0x00000001,
    SD_HOOK_ROUTINE_MSK = 0x0000000e,
    SD_HOOK_CREATE      = 0x00000002,
    SD_HOOK_OPEN        = 0x00000004,
    SD_HOOK_DELETE      = 0x00000006,
    SD_HOOK_MAP         = 0x00000008,
    SD_HOOK_UNMAP       = 0x0000000a,
    SD_HOOK_PROTECT     = 0x0000000c
    } SD_HOOK_OPTIONS;
```

Only **sdCreate( )** and **sdOpen( )** invoke the generic hook with the **SD_HOOK_NAME** option specified.

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (i.e. it will be the first hook to execute).

Shared Data generic hooks are called from **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, and **sdProtect( )** and should return either **OK** or **ERROR**. If the return value from a generic hook is anything other than **OK** the operation is aborted and the routine from which it was invoked returns **ERROR**.

RETURNS **OK**, or **ERROR** if the table of SD create routines is full.

ERRNO N/A.

SEE ALSO **sdLib**, **sdGenericHookDelete( )**

sdGenericHookDelete( )

NAME **sdGenericHookDelete( )** – delete a Shared Data generic hook routine

SYNOPSIS

```
STATUS sdGenericHookDelete
    (
    SD_GENERIC_HOOK sdGenericHook  /* hook routine to delete */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of Shared Data generic hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **sdLib**, **sdGenericHookAdd( )**

sdInfoGet( )

NAME

sdInfoGet( ) – get specific information about a Shared Data Region

SYNOPSIS

```
STATUS sdInfoGet
    (
    SD_ID     sdId,      /* SD ID to get info */
    SD_DESC * pSdStruct  /* location to store SD info */
    )
```

DESCRIPTION

This routine obtains the information for a Shared Data region and stores the information in the specified SD descriptor (sdStruct). The information stored in the descriptor is copied from information in the SD object. The descriptor must have been allocated before calling this function, and the memory for it must come from the calling task's RTP space. To allocate the memory for the descriptor from the calling task's RTP space, either use **malloc( )** within the calling task or declare the structure as an automatic variable in the calling task, placing it on the calling task's stack.

If the name of the Shared Data region is longer than **VX_SD_NAME_LENGTH** characters it will be truncated.

The sdStruct structure looks like the following:

```
typedef struct
   {
   char        name[VX_SD_NAME_LENGTH+1]; // name of SD
   int         options;        // options, e.g. SD_LINGER, SD_PRIVATE
   MMU_ATTR    defaultAttr;    // default attributes of SD
   MMU_ATTR    currentAttr;    // current attributes of SD
   UINT        size;           // size of SD in bytes
   VIRT_ADDR   startAddr       // start address of SD
   } SD_DESC;
```

See the header file **vmLibCommon.h** for definitions of the values returned in *defaultAttr* and *currentAttr*.

RETURNS

OK, or **ERROR** on failure.

ERRNO

Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

SEE ALSO

sdLib, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

sdMap( )

NAME

sdMap( ) – map a shared data region into an application or the kernel

SYNOPSIS

```
VIRT_ADDR sdMap
    (
    SD_ID    sdId,    /* ID of shared data region to map */
    MMU_ATTR attr,    /* MMU attr used to map region */
    int      options  /* reserved - use zero */
    )
```

DESCRIPTION

This routine maps the shared data region specified by *sdId* into the current calling task's memory context. The region is then available to all tasks within that application, or all tasks in the kernel if the calling task was a kernel task.

The shared data region is mapped using the MMU attributes specified by *attr*. These attributes must be equal to, or a subset of the default attributes of *sdId*. If 0 was passed then the default attributes of *sdId* are used. It is possible to use this routine to set the attributes on a shared data region for the calling task's RTP even if *sdId* is currently mapped in its memory context.

Basic MMU attribute definitions for shared data regions are provided in the **sdLibCommon.h** header file. These include:

Attribute	Meaning
SD_ATTR_RW	Read/Write for both Supervisor and User Modes
SD_ATTR_RO	Read Only for both Supervisor and User Modes
SD_ATTR_RWX	Read/Write/Execute for both Supervisor and User Modes
SD_ATTR_RX	Read/Execute for both Supervisor and User Modes

If more specific MMU attributes are required please see **vmLibCommon.h** for a complete list of available MMU attributes.

NOTE

The **MMU_ATTR** mask used internally by the shared data library is the combination of:

MMU_ATTR_PROT_MASK

MMU_ATTR_VALID_MSK

MMU_ATTR_SPL_MSK

Care must be taken to provide suitable values for all these attributes.

There are currently no options specified for this function, zero should be passed in the options parameter.

RETURNS

The base virtual address of the shared data region, or **NULL** on failure.

ERRNO

Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

S_sdLib_SD_IS_PRIVATE
sdId is private to another application

SEE ALSO **sdLib**, **sdCreate( )**, **sdOpen( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

sdOpen( )

NAME

sdOpen( ) – open a shared data region for use

SYNOPSIS

```
SD_ID sdOpen
    (
    char *   name,         /* name of SD to open or create */
    int      options,      /* open options */
    int      mode,         /* open mode */
    UINT32   size,         /* size of shared data in bytes */
    off_t64  physAddress,  /* optional physical address */
    MMU_ATTR attr,         /* allowed MMU attributes */
    void **  pVirtAddress  /* virtual return address */
    )
```

DESCRIPTION

This routine takes a shared data region name and looks for the region in the system. If the region does not exist in the system, and the **OM_CREATE** flag is specified in *mode*, then a new shared data region is created and mapped to the application. If *mode* does not specify **OM_CREATE** then no shared data region is created and **NULL** is returned. If the region does already exist in the system it is mapped into the calling task's memory context.

The following table shows each parameter and whether it is required or not:

Parameter	Required?	Default
name	Yes	N/A
options	No	0
mode	No	0
size	Yes	N/A
physAddress	No	System Allocated
attr	No	Read/Write, System Default Cache Setting
pVirtAddress	Yes	N/A

If the region specified by *name* already exists in the system all other arguments, except *pVirtAddress* and *attr* (if specified) will be ignored. In this case the region will be mapped into the calling task's memory context and the start address of the region will still be stored at *pVirtAddress* and the **SD_ID** of the region will be returned.

Currently there are only two possible values of *options*:

Option name	Value	Meaning
SD_LINGER	0x1	SD region may remain after the last client unmaps.
SD_PRIVATE	0x2	SD region is only available in the owner RTP.

Currently there are only two possible values of *mode* other than the default (0):

Mode	Meaning
DEFAULT (0)	Do not create an SD region if a matching name was not found.
OM_CREATE	Create a shared data region if a matching name was not found.
OM_EXCL	When set jointly with **OM_CREATE**, create a new shared data region immediately without attempting to open an existing shared data region. An error condition is returned if a shared data region with *name* already exists. This attribute has no effect if the **OM_CREATE** attribute is not specified.

The value of *size* must be greater than 0. It is rounded up to a page aligned size determined by the architecture.

If *physAddress* is specified and the address is not available, **NULL** will be returned. The *physAddress* specified must be aligned on the architecture dependent page size boundary and must not be mapped to any other memory context.

The MMU attributes specified in *attr* will be used as the default attributes of the shared data region. All client applications will use these by default, and may only change the local access permissions to a subset of these. The application which creates the region will have read and write access in addition to the defaults and will be allowed to set local permissions to any allowed by the architecture.

Basic MMU attribute definitions for shared data regions are provided in the **sdLibCommon.h** header file. These include:

Attribute	Meaning
SD_ATTR_RW	Read/Write for both Supervisor and User Modes
SD_ATTR_RO	Read Only for both Supervisor and User Modes
SD_ATTR_RWX	Read/Write/Execute for both Supervisor and User Modes
SD_ATTR_RX	Read/Execute for both Supervisor and User Modes
SD_CACHE_COPYBACK	Copyback cache mode
SD_CACHE_WRITETHROUGH	Write through cache mode
SD_CACHE_OFF	Cache Off

One of each the **SD_ATTR** and **SD_CACHE** macros above must be provided. The **SD_CACHE** macros can not be combined.

The cache attributes of a shared data region can not be changed after creation. All clients of that region will use the value provided at create time, including the owner.

If more specific MMU attributes are required please see **vmLibCommon.h** for a complete list of available MMU attributes.

NOTE

The **MMU_ATTR** mask used internally by the shared data library is the combination of:

MMU_ATTR_PROT_MASK

MMU_ATTR_VALID_MSK

MMU_ATTR_SPL_MSK

Care must be taken to provide suitable values for all these attributes.

The start address of the shared data region is stored at the location specified by *pVirtAddress*. This must be a valid address within the context of the calling application. It can not be **NULL**.

The **SD_ID** returned is private to the calling application. It can be shared between tasks within that application but not with tasks that reside outside that application.

RETURNS

SD_ID of opened Shared Data region, or **NULL** on failure.

ERRNO

Possible errno values set by this routine are:

S_sdLib_VIRT_ADDR_PTR_IS_NULL
pVirtAddress is **NULL**

S_sdLib_ADDR_NOT_ALIGNED
physAddress is not properly aligned

S_sdLib_PHYS_ADDR_OUT_OF_RANGE
physAddress exceeds physical address space

S_sdLib_SIZE_IS_NULL
size is **NULL**

S_sdLib_INVALID_OPTIONS
options is not a valid combination

S_sdLib_VIRT_PAGES_NOT_AVAILABLE
not enough virtual space left in system

S_sdLib_PHYS_PAGES_NOT_AVAILABLE
not enough physical memory left in system

SEE ALSO

sdLib, **sdCreate( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

sdProtect( )

NAME

sdProtect( ) – change the protection attributes of a mapped SD

SYNOPSIS

```
STATUS sdProtect
```

```
(
SD_ID    sdId,  /* ID of shared data region */
MMU_ATTR attr   /* new attributes to set */
)
```

DESCRIPTION

This routine allows the caller to change the protection of a mapped shared data region in its memory context. The shared data must be mapped in the context of the calling task.

These attributes must be equal to, or a subset of the default attributes of *sdId*. If 0 was passed then the default attributes of *sdId* are used.

The default attributes of *sdId* may be retrieved by calling the routine **sdInfoGet( )**.

Basic MMU attribute definitions for shared data regions are provided in the **sdLibCommon.h** header file. These include:

Attribute	Meaning
SD_ATTR_RW	Read/Write for both Supervisor and User Modes
SD_ATTR_RO	Read Only for both Supervisor and User Modes
SD_ATTR_RWX	Read/Write/Execute for both Supervisor and User Modes
SD_ATTR_RX	Read/Execute for both Supervisor and User Modes

NOTE

The **MMU_ATTR** mask used internally by the shared data library is the combination of:

MMU_ATTR_PROT_MASK

MMU_ATTR_VALID_MSK

MMU_ATTR_CACHE_MSK

MMU_ATTR_SPL_MSK

Care must be taken to provide suitable values for all these attributes.

RETURNS

OK, or **ERROR** on failure.

ERRNO

Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

S_sdLib_NOT_MAPPED
sdId is not mapped to the current application

SEE ALSO

sdLib, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, **sdDelete( )**

sdShow( )

NAME

sdShow( ) – display information for shared data regions

SYNOPSIS

```
BOOL sdShow
    (
    char * sdNameOrId,  /* SD name or ID */
    int    level        /* 0 = summary, 1 = detailed, 2 = all */
    )
```

DESCRIPTION

This routine displays information for a shared data region. This routine takes two parameters, *sdNameOrId* and *level*. The first parameter can either be an SD ID or an SD name string. The second parameter is the level of detail to display the information for the SDs.

Depending on the level and the SD ID specified, the information displayed differs. If the *level* is 0, then it displays the summary information for either the specified SD or all SDs in the system. If the *level* is 1, then **sdShow( )** displays the detailed information, including the client information, for the specified SD or all SDs in the system (if **SD_ID** is **NULL**). If *level* is 2, **sdShow( )** displays the detailed information for all SDs in the system, regardless of the SD ID you specify. Refer to the table for more information.

Level	SD Name or ID	Meaning
0	0	Display summary information for all SDs.
0	SD	Display summary information for specified SD.
1	0	Display detailed information for all SDs.
1	SD	Display detailed information for specified SD.
2	ANY	Display detailed information for all SD.

sdShow( ) only displays the SD name up to a maximum of 12 characters long. If the name is more than 12 characters, the name will be truncated to 10 characters for displaying purposes. Following the truncated name, a ">" will be display to indicate that the name is more than 12 characters long. To get a display of the full SD name, display the SD with the *level* set to 1.

SUMMARY INFORMATION EXAMPLE

The following example shows the summary output for all SDs in the system. If a SD ID (or name) is specified, only the information for that SD will be displayed.

```
-> sdShow

    NAME         ID      VIRT ADDR      PHYS ADDR         SIZE    CLIENT CNT
------------ ---------- ---------- ------------------ ---------- ----------
mySharedDa >   0x4c1820 0xa0000000         0x017fa000     0x1000          1

value = 0 = 0x0
```

The display contains the following fields:

Field	Meaning
NAME	The name of the SD.
ID	The numeric ID associated with the SD in the kernel.
VIRT ADRS	The virtual start address of the SD.
PHYS ADRS	The physical start address of the SD.
SIZE	SD size in bytes.
CLIENT CNT	Number of clients of the SD.

DETAILED INFORMATION EXAMPLE

The following example shows the detailed output for a single SD (i.e. the level was specified as 1). If the level is specified as 2, the detailed information is displayed for all SDs in the system and the user is prompted to press **return** or **Q** between each SD.

```
  -> sdShow 0x4c1820, 1

    NAME         ID     VIRT ADDR      PHYS ADDR         SIZE    CLIENT CNT
------------ ---------- ---------- ------------------ ---------- ----------
mySharedDa >   0x4c1820 0xa0000000         0x017fa000     0x1000          1

Full Name:          mySharedDataRegion
Options (0x1):      SD_LINGER

Default MMU Attributes (0x85b):

      ACCESS                  CACHE
     ----------------------- ---------
     Sup.: RW-  User: RW-    DEFAULT

Clients:

      NAME         ID          ACCESS                  CACHE
     ------------ ----------  ----------------------- ----------
     kernel         0x25a7c0  Sup.: RWX  User: RWX    CB /--/--

value = 0 = 0x0
```

The summary line contains the same fields as explained above. The additional information is explained in the following table:

Field	Meaning
Full Name	The complete name for the SD.
Options	Detailed breakdown of the options word (see **sdCreate( )**).
Default MMU Attributes	Default MMU attributes for the SD.
Clients	Complete list of current clients and their access rights.

RETURNS N/A

ERRNOS

Possible errnos generated by this function include:

S_objLib_OBJ_ID_ERROR
An incorrect SD ID was provided.

SEE ALSO

sdShow, **sdLib**, **rtpLib**, **vmBaseLib**, the VxWorks programmer guides.

sdUnmap( )

NAME

sdUnmap( ) – unmap a shared data region from an application or the kernel

SYNOPSIS

```
STATUS sdUnmap
    (
    SD_ID sdId,    /* ID of shared data region to unmap */
    int   options  /* options */
    )
```

DESCRIPTION

This routine unmaps the shared data region specified by *sdId* from the calling task's memory context. The region is then no longer available to any tasks within that application, or any tasks in the kernel if the calling task was a kernel task. There are currently no options specified for this function, zero should be passed in the options parameter.

RETURNS

OK, or **ERROR** on failure.

ERRNO

Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

S_sdLib_NOT_MAPPED
sdId is not mapped to the current application

SEE ALSO

sdLib, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdProtect( )**, **sdDelete( )**

selNodeAdd( )

NAME

selNodeAdd( ) – add a wake-up node to a **select( )** wake-up list

SYNOPSIS

```
STATUS selNodeAdd
    (
    SEL_WAKEUP_LIST *pWakeupList,  /* list of tasks to wake up */
    SEL_WAKEUP_NODE *pWakeupNode   /* node to add to list */
    )
```

DESCRIPTION This routine adds a wake-up node to a device's wake-up list. It is typically called from a driver's FIOSELECT function.

RETURNS **OK**, or **ERROR** if memory is insufficient.

ERRNO N/A

SEE ALSO **selectLib**

selNodeDelete( )

NAME **selNodeDelete( )** – find and delete a node from a **select( )** wake-up list

SYNOPSIS

```
STATUS selNodeDelete
    (
    SEL_WAKEUP_LIST *pWakeupList,  /* list of tasks to wake up */
    SEL_WAKEUP_NODE *pWakeupNode   /* node to delete from list */
    )
```

DESCRIPTION This routine deletes a specified wake-up node from a specified wake-up list. Typically, it is called by a driver's FIOUNSELECT function.

RETURNS **OK**, or **ERROR** if the node is not found in the wake-up list.

ERRNO N/A

SEE ALSO **selectLib**

selWakeup( )

NAME **selWakeup( )** – wake up a task pended in **select( )**

SYNOPSIS

```
void selWakeup
    (
    SEL_WAKEUP_NODE *pWakeupNode  /* node to wake up */
    )
```

DESCRIPTION This routine wakes up a task pended in **select( )**. Once a driver's FIOSELECT function installs a wake-up node in a device's wake-up list (using **selNodeAdd( )**) and checks to make sure the device is ready, this routine ensures that the **select( )** call does not pend.

RETURNS N/A

ERRNO N/A

SEE ALSO **selectLib**

selWakeupAll( )

NAME **selWakeupAll( )** – wake up all tasks in a **select( )** wake-up list

SYNOPSIS

```
void selWakeupAll
    (
    SEL_WAKEUP_LIST  *pWakeupList,  /* list of tasks to wake up */
    FAST SELECT_TYPE type           /* readers (SELREAD) or writers
(SELWRITE) */
    )
```

DESCRIPTION This routine wakes up all tasks pended in **select( )** that are waiting for a device; it is called by a driver when the device becomes ready. The *type* parameter specifies the task to be awakened, either reader tasks (SELREAD) or writer tasks (SELWRITE).

RETURNS N/A

ERRNO N/A

SEE ALSO **selectLib**

selWakeupListInit( )

NAME **selWakeupListInit( )** – initialize a **select( )** wake-up list

SYNOPSIS

```
void selWakeupListInit
    (
    SEL_WAKEUP_LIST *pWakeupList  /* wake-up list to initialize */
    )
```

DESCRIPTION This routine should be called in a device's create routine to initialize the **SEL_WAKEUP_LIST** structure.

RETURNS N/A

ERRNO N/A

SEE ALSO **selectLib**

selWakeupListLen( )

NAME **selWakeupListLen( )** – get the number of nodes in a **select( )** wake-up list

SYNOPSIS

```
int selWakeupListLen
    (
    SEL_WAKEUP_LIST *pWakeupList  /* list of tasks to wake up */
    )
```

DESCRIPTION This routine returns the number of nodes in a specified **SEL_WAKEUP_LIST**. It can be used by a driver to determine if any tasks are currently pended in **select( )** on this device, and whether these tasks need to be activated with **selWakeupAll( )**.

RETURNS The number of nodes currently in a **select( )** wake-up list, or **ERROR**.

ERRNO N/A

SEE ALSO **selectLib**

selWakeupListTerm( )

NAME **selWakeupListTerm( )** – terminate a **select( )** wake-up list

SYNOPSIS

```
void selWakeupListTerm
    (
    SEL_WAKEUP_LIST *pWakeupList  /* wake-up list to terminate */
    )
```

DESCRIPTION This routine should be called in a device's terminate routine to terminate the **SEL_WAKEUP_LIST** structure.

RETURNS N/A

ERRNO N/A

SEE ALSO **selectLib**

selWakeupType( )

NAME **selWakeupType( )** – get the type of a **select( )** wake-up node

SYNOPSIS

```
SELECT_TYPE selWakeupType
    (
    SEL_WAKEUP_NODE *pWakeupNode  /* node to get type of */
    )
```

DESCRIPTION This routine returns the type of a specified **SEL_WAKEUP_NODE**. It is typically used in a device's FIOSELECT function to determine if the device is being selected for read or write operations.

RETURNS SELREAD (read operation) or SELWRITE (write operation).

ERRNO N/A

SEE ALSO **selectLib**

select( )

NAME **select( )** – pend on a set of file descriptors

SYNOPSIS

```
int select
    (
    int             width,       /* number of bits to examine from 0 */
    FAST fd_set    *pReadFds,   /* read fds */
    FAST fd_set    *pWriteFds,  /* write fds */
    fd_set         *pExcFds,    /* exception fds */
    struct timeval *pTimeOut    /* max time to wait, NULL = forever */
    )
```

DESCRIPTION This routine permits a task to pend until one of a set of file descriptors becomes ready. Three parameters -- *pReadFds*, *pWriteFds*, and *pExceptFds* -- point to file descriptor sets in which each bit corresponds to a particular file descriptor. Bits set in the read file descriptor set (*pReadFds*) will cause **select( )** to pend until data is available on any of the corresponding file descriptors, while bits set in the write file descriptor set (*pWriteFds*) will cause **select( )** to pend until any of the corresponding file descriptors become writable.

The following macros are available for setting the appropriate bits in the file descriptor set structure:

```
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ZERO(&fdset)
```

If either *pReadFds*, *pWriteFds*, or *pExceptFds* is **NULL**, they are ignored. The *width* parameter defines how many bits will be examined in the file descriptor sets, and should be set to either the maximum file descriptor value in use plus one, or simply to **FD_SETSIZE**. When **select( )** returns, it zeros out the file descriptor sets, and sets only the bits that correspond to file descriptors that are ready. The **FD_ISSET** macro may be used to determine which bits are set.

If *pTimeOut* is **NULL**, **select( )** will block indefinitely. If *pTimeOut* is not **NULL**, but points to a **timeval** structure with an effective time of zero, the file descriptors in the file descriptor sets will be polled and the results returned immediately. If the effective time value is greater than zero, **select( )** will return after the specified time has elapsed, even if none of the file descriptors are ready.

Applications can use **select( )** with pipes and serial devices, in addition to sockets. Select now has the capability to support exception reports, but note that most devices do not provide exception notification for select activity. Refer to the manual for each particular driver to learn about its **select( )** support, if any.

The value for the maximum number of file descriptors configured in the system (**NUM_FILES**) should be less than or equal to the value of **FD_SETSIZE** (2048).

Driver developers should consult the VxWorks programmer guides for details on writing drivers that will use **select( )**.

RETURNS

The number of file descriptors with activity, 0 if timed out, or **ERROR** if an error occurred when the driver's **select( )** routine was invoked via **ioctl( )**.

ERRNOS

Possible errnos generated by this routine include:

S_selectLib_NO_SELECT_SUPPORT_IN_DRIVER
A driver associated with one or more fds does not support **select( )**.

S_selectLib_NO_SELECT_CONTEXT
The task's select context was not initialized at task creation time.

S_selectLib_WIDTH_OUT_OF_RANGE
The width parameter is greater than the maximum possible fd.

S_memLib_NOT_ENOUGH_MEMORY
Heap allocation failure has caused select to fail.

EBADF
An invalid file descriptor was specified in one of the sets, or a valid file descriptor which was specified was closed by another task while the **select( )** call was in progress. (Note, closing a file descriptor in use by another task is NOT recommended.)

SEE ALSO **selectLib**, the VxWorks programmer guides.

selectInit( )

NAME **selectInit( )** – initialize the select facility

SYNOPSIS

```
void selectInit
    (
    int numFiles  /* no longer used */
    )
```

DESCRIPTION This routine initializes the UNIX BSD 4.3 select facility. It is initialized automatically when the **INCLUDE_SELECT** component is configured. It installs a task create hook such that a select context is initialized for each task.

RETURNS N/A

ERRNO N/A

SEE ALSO **selectLib**

semBCreate( )

NAME **semBCreate( )** – create and initialize a binary semaphore

SYNOPSIS

```
SEM_ID semBCreate
    (
    int         options,      /* semaphore options */
    SEM_B_STATE initialState  /* initial semaphore state */
    )
```

DESCRIPTION This routine allocates and initializes a binary semaphore. The semaphore is initialized to the *initialState* of either **SEM_FULL** (1) or **SEM_EMPTY** (0).

The *options* parameter specifies the queuing style blocked tasks and response on signals for blocked RTP tasks. Tasks may be queued on a priority basis or a first-in-first-out basis. The queuing style options are **SEM_Q_PRIORITY** (0x1) and **SEM_Q_FIFO** (0x0), respectively. That parameter also specifies if **semGive( )** should return **ERROR** when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of **SEM_EVENTSEND_ERR_NOTIFY** (0x10) with the queuing style of the semaphore. **SEM_INTERRUPTIBLE**(0x20) is the option which makes the blocked RTP task on the semaphore ready and return **ERROR** with errno set to **EINTR** when a signal is generated to that task. This option has no affect when a kernel task blocks on the same semaphore created with this option. This option is turned off by default.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. This restriction is not strictly enforced.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated or if error.

ERRNO

S_semLib_INVALID_OPTION
Invalid option was specified.

S_memLib_NOT_ENOUGH_MEMORY
Not enough memory available to create the semaphore.

S_semLib_INVALID_STATE
Invalid initial state.

S_semLib_INVALID_QUEUE_TYPE
Invalid type of semaphore queue specified.

S_spinLockLib_NOT_SPIN_LOCK_CALLABLE
This API is spinlock restricted and can not be called taking a spinlock.

SEE ALSO

semBLib, **semLib**

semBInitialize( )

NAME

semBInitialize( ) – initialize a pre-allocated binary semaphore.

SYNOPSIS

```
SEM_ID semBInitialize
    (
    char *      pSemMem,      /* pointer to allocated storage */
    int         options,      /* semaphore options */
    SEM_B_STATE initialState  /* initial semaphore state */
    )
```

DESCRIPTION

This routine initializes a binary semaphore that has been pre-allocated (i.e. by the **VX_BINARY_SEMAPHORE** macro). The semaphore is initialized and an ID is returned for further operations on this semaphore.

The *options* and *initialState* parameters have the same meaning as those for **semBCreate( )**. Please see the documentation for **semBCreate( )** for more details.

The following example illustrates use of the **VX_BINARY_** SEMAPHORE macro and this function together to instantiate a binary semaphore statically (without using any dynamic memory allocation):

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

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

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

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. This restriction is not strictly enforced.

RETURNS The semaphore ID, or **NULL** on error.

ERRNO **S_spinLockLib_NOT_SPIN_LOCK_CALLABLE**
This API is spinlock restricted and can not be called taking a spinlock.

SEE ALSO **semBLib**

semBSmCreate( )

NAME **semBSmCreate( )** – create and initialize a shared memory binary semaphore (VxMP Option)

SYNOPSIS

```
SEM_ID semBSmCreate
    (
    int         options,      /* semaphore options */
    SEM_B_STATE initialState  /* initial semaphore state */
    )
```

DESCRIPTION This routine allocates and initializes a shared memory binary semaphore. The semaphore is initialized to an *initialState* of either **SEM_FULL** (available) or **SEM_EMPTY** (not available). The shared semaphore structure is allocated from the shared semaphore dedicated memory partition.

The semaphore ID returned by this routine can be used directly by the generic semaphore-handling routines in **semLib** -- **semGive( )**, **semTake( )**, and **semFlush( )** -- and the show routines, such as **show( )** and **semShow( )**.

The queuing style for blocked tasks is set by *options*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO** .

Before this routine can be called, the shared memory objects facility must be initialized (see **semSmLib**).

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM** , a configurable parameter.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS The semaphore ID, or **NULL** if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine has been called from ISR.

S_objLib_OBJ_ID_ERROR
The shared memory semaphore partition has not been initialized properly.

S_memLib_NOT_ENOUGH_MEMORY
Can't allocate shared memory semaphore object.

S_semLib_INVALID_QUEUE_TYPE
Incorrect semaphore pend queue type specified.

S_semLib_INVALID_STATE
Incorrect initial semaphore state specified.

S_smObjLib_LOCK_TIMEOUT
Can't get the lock on the shared memory semaphore partition in time.

SEE ALSO **semSmLib**, **semLib**, **semBLib**, **smObjLib**, **semShow**, the VxWorks programmer guides.

semCCreate( )

NAME **semCCreate( )** – create and initialize a counting semaphore

SYNOPSIS

```
SEM_ID semCCreate
    (
    int options,      /* semaphore option modes */
    int initialCount  /* initial count */
    )
```

DESCRIPTION

This routine allocates and initializes a counting semaphore. The semaphore is initialized to the specified initial count.

The *options* parameter specifies the queuing style and response on signals for blocked RTP tasks. Tasks may be queued on a priority basis or a first-in-first-out basis. The queuing style options are **SEM_Q_PRIORITY** (0x1) and **SEM_Q_FIFO** (0x0), respectively. That parameter also specifies if **semGive( )** should return **ERROR** when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of **SEM_EVENTSEND_ERR_NOTIFY** (0x10) with the queuing style of the semaphore. **SEM_INTERRUPTIBLE**(0x20) is the option which makes the blocked RTP task on the semaphore ready and return **ERROR** with errno set to **EINTR** when a signal is generated to that task. This option has no affect when a kernel task blocks on the same semaphore created with this option. This option is turned off by default.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated or error.

ERRNO

S_semLib_INVALID_INITIAL_COUNT
The specified initial count is negative

S_semLib_INVALID_OPTION
Options not applicable to counting semaphores were specified.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to create the semaphore.

SEE ALSO

semCLib, **semLib**

semCInitialize( )

NAME

semCInitialize( ) – initialize a pre-allocated counting semaphore.

SYNOPSIS

```
SEM_ID semCInitialize
    (
    char * pSemMem,      /* pointer to allocated storage */
    int    options,      /* semaphore options */
    int    initialCount  /* initial count */
    )
```

DESCRIPTION

This routine initializes a counting semaphore that has been pre-allocated (i.e. by the **VX_COUNTING_SEMAPHORE** macro). The semaphore is initialized and an ID is returned for further operations on this semaphore.

The *options* and *initialCount* parameters have the same meaning as those for **semCCreate( )**. Please see the documentation for **semCCreate( )** for more details.

The following example illustrates use of the **VX_COUNTING_SEMAPHORE** macro and this function together to instantiate a counting semaphore statically (without using any dynamic memory allocation):

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

VX_COUNTING_SEMAPHORE(mySemC); /* declare the semaphore */
SEM_ID mySemCId;               /* semaphore ID for further operations */

STATUS initializeFunction (void)
    {
    if ((mySemCId = semCInitialize (mysemC, options, 0)) == NULL)
         return (ERROR);       /* initialization failed */
    else
         return (OK);
    }
```

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

The semaphore ID, or **NULL** on error.

ERRNO

N/A

SEE ALSO

semCLib

semCSmCreate( )

NAME

semCSmCreate( ) – create and initialize a shared memory counting semaphore (VxMP Option)

SYNOPSIS

```
SEM_ID semCSmCreate
    (
    int options,      /* semaphore options */
    int initialCount  /* initial semaphore count */
    )
```

DESCRIPTION

This routine allocates and initializes a shared memory counting semaphore. The initial count value of the semaphore is specified by *initialCount*.

The semaphore ID returned by this routine can be used directly by the generic semaphore-handling routines in **semLib** -- **semGive( )**, **semTake( )** and **semFlush( )** -- and the show routines, such as **show( )** and **semShow( )**.

The queuing style for blocked tasks is set by *options*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO** .

Before this routine can be called, the shared memory objects facility must be initialized (see **semSmLib**).

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM** , a configurable paramter.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine has been called from ISR.

S_objLib_OBJ_ID_ERROR
The shared memory semaphore partition has not been initialized properly.

S_memLib_NOT_ENOUGH_MEMORY
Can't allocate shared memory semaphore object.

S_semLib_INVALID_QUEUE_TYPE
Incorrect semaphore pend queue type specified.

S_semLib_INVALID_COUNT
Incorrect initial count (negative) specified.

S_smObjLib_LOCK_TIMEOUT
Can't get the lock on the shared memory semaphore partition in time.

SEE ALSO

semSmLib, **semLib**, **semCLib**, **smObjLib**, **semShow**, the VxWorks programmer guides.

semClose( )

NAME

semClose( ) – close a named semaphore

SYNOPSIS

```
STATUS semClose
```

```
    (
    SEM_ID semId  /* semaphore ID to close */
    )
```

DESCRIPTION

This routine closes a named semaphore. It decrements the semaphore's reference counter. In case it becomes zero, the semaphore is deleted if:

- It has been already removed from the name space by a call to **semUnlink( )**.
- It was created with the **OM_DESTROY_ON_LAST_CLOSE** option.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_INVALID_ARGUMENT
Semaphore ID is **NULL**.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Semaphore is not named.

S_objLib_OBJ_DESTROY_ERROR
Error while deleting the semaphore.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO

semOpen, semOpen, semUnlink

semDelete( )

NAME

semDelete( ) – delete a semaphore

SYNOPSIS

```
STATUS semDelete
    (
    SEM_ID semId  /* semaphore ID to delete */
    )
```

DESCRIPTION
This routine terminates and deallocates any memory associated with a specified semaphore. All tasks pending on the semaphore or pending for the reception of events meant to be sent from the semaphore will unblock and return **ERROR**.

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted.

WARNING
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.

RETURNS
OK, or **ERROR** if the semaphore ID is invalid.

ERRNOS

S_intLib_NOT_ISR_CALLABLE
Routine cannot be called from ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_smObjLib_NO_OBJECT_DESTROY
Deleting a shared semaphore is not permitted

S_objLib_OBJ_OPERATION_UNSUPPORTED
Deleting a named semaphore is not permitted.

SEE ALSO
semLib, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

semEvStart( )

NAME
semEvStart( ) – start the event notification process for a semaphore

SYNOPSIS
```
STATUS semEvStart
    (
    SEM_ID semId,   /* semaphore on which to register events */
    UINT32 events,  /* 32 possible events to register        */
    UINT8  options  /* event-related semaphore options       */
    )
```

DESCRIPTION
This routine turns on the event notification process for a given semaphore, registering the calling task on that semaphore. When the semaphore becomes available but no task is pending on it, the events specified will be sent to the registered task. A task can always overwrite its own registration.

The *events* are user-defined. For more information, see the reference entry for **eventLib**.

The *option* parameter is used for 3 user options:

- Specify if the events are to be sent only once or every time the semaphore becomes free until **semEvStop( )** is called.
- Specify if another task can subsequently register itself while the calling task is still registered. If so specified, the existing task registration will be overwritten without any warning.
- Specify if events are to be sent at the time of the registration in the case the semaphore is free.

Here are the respective values to be used to form the options field:

EVENTS_SEND_ONCE (0x1)
The semaphore will send the events only once.

EVENTS_ALLOW_OVERWRITE (0x2)
Subsequent registrations from other tasks may overwrite the current one.

EVENTS_SEND_IF_FREE (0x4)
The registration process will send events if the semaphore is free at the time **semEvStart( )** is called.

EVENTS_OPTIONS_NONE (0x0)
Must be passed to the *options* parameter if none of the other three options are used.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

WARNING

This routine cannot be called from interrupt level.

WARNING

Task preemption can allow a **semDelete( )** to be performed between the calls to **semEvStart( )** and **eventReceive( )**. This prevents the task from ever receiving the events wanted from the semaphore.

RETURNS

OK on success, or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
The semaphore ID is invalid.

S_eventLib_ALREADY_REGISTERED
A task is already registered on the semaphore.

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from interrupt level.

S_eventLib_EVENTSEND_FAILED
The user chose to send events immediately and that operation failed.

S_eventLib_ZERO_EVENTS
The user passed in a value of zero to the *events* parameter.

SEE ALSO **semEvLib**, **eventLib**, **semLib**, **semEvStop( )**

semEvStop( )

NAME **semEvStop( )** – stop the event notification process for a semaphore

SYNOPSIS

```
STATUS semEvStop
    (
    SEM_ID semId
    )
```

DESCRIPTION This routine turns off the event notification process for a given semaphore. It thus allows another task to register itself for event notification on that particular semaphore. It must be called from the task that is already registered on that particular semaphore.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** on success, or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
The semaphore ID is invalid.

S_intLib_NOT_ISR_CALLABLE
The routine cannot be called from interrupt level.

S_eventLib_TASK_NOT_REGISTERED
The routine was not called by the registered task.

SEE ALSO **semEvLib**, **eventLib**, **semLib**, **semEvStart( )**

semExchange( )

NAME **semExchange( )** – atomically give and take a pair of semaphores

SYNOPSIS

```
STATUS semExchange
    (
    SEM_ID giveSemId,  /* semaphore ID to give */
    SEM_ID takeSemId,  /* semaphore ID to take */
    int    timeout     /* timeout in ticks */
    )
```

DESCRIPTION

This routine atomically performs a give operation on a sempahore and a take operation on another semaphore. The semaphore specified to be given will be released when the caller acquires or pends attempting to acquire the semaphore specifed to be taken.

This routine performs the give operation on a semaphore specified by the *giveSemId* argument. Depending on the type of this semaphore, the state of the semaphore and of the pending tasks may be affected. If no tasks are pending on the semaphore and a task has previously registered to receive events from the semaphore, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the semaphore fails to send events and if it was created using the **SEM_EVENTSEND_ERR_NOTIFY** option, **ERROR** is returned even though the give operation was successful. The behavior of **semGive( )** is discussed fully in the library description of the specific semaphore type being used.

If the give operation returns **ERROR** for any reason the subsequent take operation will not be performed.

This routine performs the take operation on a semaphore specified by the *takeSemId* argument. Depending on the type of this semaphore, the state of the semaphore and the calling task may be affected. The behavior of **semTake( )** is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified for the **semTake( )** portion of the **semExchange( )** operation. If a task times out, **semExchange( )** will return **ERROR**. Timeouts of **WAIT_FOREVER** (-1) and **NO_WAIT** (0) indicate to wait indefinitely or not to wait at all.

When **semExchange( )** returns due to timeout, it sets the errno to **S_objLib_OBJ_TIMEOUT** (defined in **objLib.h**).

Because it completes when the caller pends during the **semTake( )** operation the **semGive( )** operation will occur regardless of timeout. It is possible for the caller to release the specified give semaphore and not acquire the semaphore specified to be taken.

The **semExchange( )** routine is not callable from interrupt service routines.

Currently only binary and mutex semaphore types are supported by **semExchange( )**.

An attempt to specify a semaphore of another type for either the give or take operation of **semExchange( )** will result in a return value of **ERROR**. Neither the give or take operation will be performed.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNOS

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_objLib_OBJ_UNAVAILABLE
Would have blocked but **NO_WAIT** was specified.

S_semLib_INVALID_OPERATION
Current task not owner of semaphore.

S_eventLib_EVENTSEND_FAILED
Semaphore failed to send events to the registered task. This errno value can only exist if the semaphore was created with the **SEM_EVENTSEND_ERR_NOTIFY** option.

SEE ALSO **semExchange**, **semLib**, **semBLib**, **semMLib**

semFlush( )

NAME **semFlush( )** – unblock every task pended on a semaphore

SYNOPSIS

```
STATUS semFlush
    (
    SEM_ID semId  /* semaphore ID to unblock everyone for */
    )
```

DESCRIPTION This routine atomically unblocks all tasks pended on a specified semaphore, i.e., all tasks will be unblocked before any is allowed to run. The state of the underlying semaphore is unchanged. All pended tasks will enter the ready queue before having a chance to execute.

The flush operation is useful as a means of broadcast in synchronization applications. Its use is illegal for mutual-exclusion semaphores created with **semMCreate( )** or with reader/writer semaphores created with **semRWCreate( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the operation is not supported.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semRWLib**, **semSmLib**

semGive( )

NAME **semGive( )** – give a semaphore

SYNOPSIS

```
STATUS semGive
    (
    SEM_ID semId  /* semaphore ID to give */
    )
```

DESCRIPTION This routine performs the give operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and of the pending tasks may be affected. If no tasks are pending on the semaphore and a task has previously registered to receive events from the semaphore, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the semaphore fails to send events and if it was created using the **SEM_EVENTSEND_ERR_NOTIFY** option, **ERROR** is returned even though the give operation was successful. The behavior of **semGive( )** is discussed fully in the library description of the specific semaphore type being used.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK** on success or **ERROR** otherwise

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR for a semaphore.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_semLib_INVALID_OPERATION
Current task not owner of semaphore.

S_semLib_COUNT_OVERFLOW
Counting semaphore was given when count was already at maximum.

S_eventLib_EVENTSEND_FAILED
Semaphore failed to send events to the registered task. This errno value can only exist if the semaphore was created with the **SEM_EVENTSEND_ERR_NOTIFY** option.

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semRWLib**, **semSmLib**, **semEvStart( )**

semInfo( )

NAME

semInfo( ) – get information about tasks blocked on a semaphore

SYNOPSIS

```
int semInfo
    (
    SEM_ID semId,      /* semaphore ID to summarize */
    int    idList[],   /* array of task IDs to be filled in */
    int    maxTasks    /* max tasks idList can accommodate */
    )
```

DESCRIPTION

This routine returns the number of tasks that are blocked on the specified semaphore, *semId*. If a non-**NULL** array is passed in *idList*, then up to *maxTasks* task IDs are copied into the array. In this case, this routine returns the number of task IDs that could be copied into the array. The array is unordered.

WARNING

There is no guarantee that all listed tasks are still valid or that new tasks have not been blocked by the time **semInfo( )** returns.

RETURNS

The actual number of blocked tasks if *idList* is **NULL**
or the number of task IDs placed in *idList*.

ERRNO

S_objLib_OBJ_ID_ERROR
Invalid semaphore ID.

S_intLib_NOT_ISR_CALLABLE
This routine is not callable from an ISR.

SEE ALSO

semInfo, **semInfoGet( )**

semInfoGet( )

NAME

semInfoGet( ) – get information about a semaphore

SYNOPSIS

```
STATUS semInfoGet
    (
    SEM_ID     semId,  /* semaphore to query */
    SEM_INFO * pInfo   /* where to return semaphore info */
    )
```

DESCRIPTION

This routine gets information about the state of a semaphore. The parameter *pInfo* is a pointer to a structure of type **SEM_INFO** defined in **semLibCommon.h** as follows:

```
typedef struct                  /* SEM_INFO */
    {
    UINT     numTasks;          /* OUT: number of blocked tasks */
    SEM_TYPE semType;           /* OUT: semaphore type */
    int      options;           /* OUT: options with which sem was created */
    union
        {
        UINT count;             /* OUT: semaphore count (counting sems) */
        BOOL full;              /* OUT: binary semaphore FULL? */
        int  owner;             /* OUT: task ID of mutex semaphore owner */
        } state;
    int      taskIdListMax;     /* IN: max tasks to fill in taskIdList */
    int *    taskIdList;        /* PTR: array of pending task IDs */
    } SEM_INFO;
```

The semaphore type is determined by examining *semType*. Based on this information the appropriate field in the *state* union can be examined to determine a) the current count of a counting semaphore *state.count*, b) whether a binary semaphore is full *state.full*, or c) the owner of a mutex semaphore *state.owner*.

If a binary semaphore is not full *state.full* = **FALSE**, or if a counting semaphore's count is 0 *state.count = 0*, or a mutex semaphore is already owned *state.owner* != **NULL**, then there may be tasks blocked on **semTake( )**. The *numTasks* field indicates the number of blocked tasks.

A list of the task IDs of tasks blocked on the semaphore can be obtained by setting *taskIdList* to the address of an array to receive the list, and setting *taskIdListMax* to the maximum number of elements in that array. If *taskIdList* is **NULL**, then no task IDs are returned. No more than *taskIdListMax* task IDs are returned, although *numTasks* will always be returned with the actual number of tasks blocked.

For example, if the caller supplies a *taskIdList* with room for 10 task IDs and sets *taskIdListMax* to 10, but there are 20 tasks blocked on the semaphore, then the IDs of the first 10 tasks blocked on the semaphore will be returned in *taskIdList*, but *numTasks* will be returned with the value 20.

The *options* field is the parameter with which the semaphore was created.

WARNING

The information returned by this routine is not static and may be obsolete by the time it is examined. In particular, the list of task IDs may no longer be valid. However, the information is obtained atomically, thus it will be an accurate snapshot of the state of the semaphore at the time of the call. This information is generally used for debugging purposes only.

If *taskIdList* is non-**NULL**, i.e. the caller is requesting the list of pended tasks, the execution time of **semInfoGet( )** may be non-deterministic.

This routine cannot be used to extract information on shared semaphores.

RETURNS

OK or **ERROR**

ERRNO

S_objLib_OBJ_ID_ERROR
Invalid semaphore ID.

S_semLib_INVALID_OPERATION
Specified semaphore is a shared semaphore, or the semaphore is an unknown type.

SEE ALSO **semInfo**, **semInfo( )**

semMCreate( )

NAME **semMCreate( )** – create and initialize a mutual-exclusion semaphore

SYNOPSIS
```
SEM_ID semMCreate
    (
    int options  /* mutex semaphore options */
    )
```

DESCRIPTION This routine allocates and initializes a mutual-exclusion semaphore. The semaphore state is initialized to full.

Semaphore options include the following:

SEM_Q_PRIORITY (0x1)
Queue pended tasks on the basis of their priority.

SEM_Q_FIFO (0x0)
Queue pended tasks on a first-in-first-out basis.

SEM_DELETE_SAFE (0x4)
Protect a task that owns the semaphore from unexpected deletion. This option enables an implicit **taskSafe( )** for each **semTake( )**, and an implicit **taskUnsafe( )** for each **semGive( )**.

SEM_INVERSION_SAFE (0x8)
Protect the system from priority inversion. With this option, the task owning the semaphore will execute at the highest priority of the tasks pended on the semaphore, if it is higher than its current priority. This option must be accompanied by the **SEM_Q_PRIORITY** queuing mode.

SEM_EVENTSEND_ERR_NOTIFY (0x10)
When the semaphore is given, if a task is registered for events and the actual sending of events fails, a value of **ERROR** is returned and the errno is set accordingly. This option is off by default.

SEM_INTERRUPTIBLE (0x20)
Signal sent to an RTP task blocked on a semaphore created with this option, would make the task ready and return with **ERROR** and errno set to **EINTR**. This option has no affect for a kernel task blocked on the same semaphore created with this option. This option is off by default.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. This restriction is not strictly enforced.

RETURNS The semaphore ID, or **NULL** if the semaphore cannot be created.

ERRNO

S_semLib_INVALID_OPTION
Invalid option was passed to semMCreate.

S_memLib_NOT_ENOUGH_MEMORY
Not enough memory available to create the semaphore.

SEE ALSO **semMLib**, **semLib**, **semBLib**, **taskSafe( )**, **taskUnsafe( )**

semMGiveForce( )

NAME **semMGiveForce( )** – give a mutual-exclusion semaphore without restrictions

SYNOPSIS

```
STATUS semMGiveForce
    (
    FAST SEM_ID semId  /* semaphore ID to give */
    )
```

DESCRIPTION This routine gives a mutual-exclusion semaphore, regardless of semaphore ownership. It is intended as a debugging aid only.

The routine is particularly useful when a task dies while holding some mutual-exclusion semaphore, because the semaphore can be resurrected. The routine will give the semaphore to the next task in the pend queue or make the semaphore full if no tasks are pending. In effect, execution will continue as if the task owning the semaphore had actually given the semaphore.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. This restriction is not strictly enforced.

CAVEATS This routine should be used only as a debugging aid, when the condition of the semaphore is known.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid.

ERRNO N/A

SEE ALSO **semMLib**, **semGive( )**

semMInitialize( )

NAME **semMInitialize( )** – initialize a pre-allocated mutex semaphore.

SYNOPSIS

```
SEM_ID semMInitialize
    (
    char * pSemMem,  /* pointer to allocated storage */
    int    options   /* mutex semaphore options */
    )
```

DESCRIPTION This routine initializes a mutual exclusion semaphore that has been pre- allocated (i.e. by the **VX_MUTEX_SEMAPHORE** macro). The semaphore is initialized and an ID is returned for further operations on this semaphore.

The *options* parameter has the same meaning as that for **semMCreate( )**. Please see the documentation for **semBCreate( )** for more details.

The following example illustrates use of the **VX_MUTEX_SEMAPHORE** macro and this function together to instantiate a mutex semaphore statically (without using any dynamic memory allocation):

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

VX_MUTEX_SEMAPHORE(mySemM);     /* declare the semaphore */
SEM_ID mySemMId;                /* semaphore ID for further operations */

STATUS initializeFunction (void)
    {
    if ((mySemMId = semMInitialize (mySemM, options, 0)) == NULL)
         return (ERROR);        /* initialization failed */
    else
         return (OK);
    }
```

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. This restriction is not strictly enforced.

RETURNS The semaphore ID, or **NULL** on error.

ERRNO **S_spinLockLib_NOT_SPIN_LOCK_CALLABLE**
This API is spinlock restricted and can not be called after taking a spinlock.

SEE ALSO **semMLib**

semOpen( )

NAME

semOpen( ) – open a named semaphore

SYNOPSIS

```
SEM_ID semOpen
    (
    const char * name,       /* name of semaphore */
    SEM_TYPE     type,       /* type of semaphore */
    int          initState,  /* initial state or initial count */
    int          options,    /* semaphore options */
    int          mode,       /* OM_CREATE, ... */
    void *       context     /* context value */
    )
```

DESCRIPTION

This routine either opens an existing semaphore or creates a new semaphore if the appropriate flags in the *mode* parameter are set. A semaphore with the name specified by the *name* parameter is searched for, and if found the **SEM_ID** of the semaphore is returned. A new semaphore may only be created if the search of existing semaphores fails (ie. the name must be unique).

There are two name spaces in which **semOpen( )** can perform a search in, the "private to the application" name space and the "public" name space. Which is selected depends on the first character in the *name* parameter. When this character is a forward slash **/**, the "public" name space is used, otherwise the the "private to the application" name space is used.

Semaphores created by this routine can not be deleted with **semDelete( )**. Instead, a **semClose( )** must be issued for every **semOpen( )**. Then the semaphore is deleted when it is removed from the name space by a call to **semUnlink( )**. Alternatively, the semaphore can be previously removed from the name space, and deleted during the last **semClose( )**.

The parameters to the semOpen function are as follows:

name
A mandatory text string which represents the name by which the semaphore is known by. **NULL** or empty strings can not be used.

type
When creating a semaphore, it specifies which type of semaphore is to be created. The valid types are:

SEM_TYPE_BINARY	create a binary semaphore
SEM_TYPE_MUTEX	create a mutual exclusion semaphore
SEM_TYPE_COUNTING	create a counting semaphore

initState
When a binary or counting semaphore is created, the initial state of the semaphore is set according to the value of *initState*. For binary semaphores the value of *initState* must be either **SEM_FULL** or **SEM_EMPTY**. For counting semaphores the semaphore count is set to the value of *initState*.

options
Semaphore creation options as decribed in **semLib**.

mode
The mode parameter consists of the access rights (which are currently ignored) and the opening flags which are bitwise-OR'd together. The flags available are:

OM_CREATE
Create a new semaphore if a matching semaphore name is not found.

OM_EXCL
When set jointly with the **OM_CREATE** flag, creates a new semaphore immediately without trying to open an existing semaphore. The call fails if the semaphore's name causes a name clash. This flag has no effect if the **OM_CREATE** flag is not specified.

OM_DELETE_ON_LAST_CLOSE
Only used when a semaphore is created. If set, the semaphore will be deleted during the last **semClose( )** call, independently on whether **semUnlink( )** was previously called or not.

context
Context value assigned to the created semaphore. This value is not actually used by VxWorks. Instead, the context value can be used by OS extensions to implement object permissions, for example.

Unlike private objects, a public semaphore is not automatically reclaimed when an application terminates. Note that nevertheless, a **semClose( )** is issued on every application's outstanding **semOpen( )**. Therefore, a public semaphore can effectively be deleted, if during this process it is closed for the last time, and it is already unlinked or it was created with the **OM_DELETE_ON_LAST_CLOSE** flag.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The **SEM_ID** of the opened semaphore, or **NULL** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid.

S_semLib_INVALID_INITIAL_COUNT
The specified initial count for counting semaphore is negative

S_objLib_OBJ_NOT_FOUND
The **OM_CREATE** flag was not set in the *mode* argument and a semaphore matching *name* was not found.

S_objLib_OBJ_NAME_CLASH
The **OM_CREATE** and **OM_EXCL** flags were set and a name clash was detected when creating the semaphore.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **semOpen**, **semUnlink( )**, **semClose( )**

semOpenInit( )

NAME **semOpenInit( )** – initialize the semaphore open facility

SYNOPSIS

```
void semOpenInit (void)
```

DESCRIPTION This routine links the semaphore creation routine with open facility into the VxWorks system. It is called automatically when the semaphore facility is configured into VxWorks by either defining **INCLUDE_OBJ_OPEN** and **INCLUDE_SEM_BINARY** in **config.h** or selecting **INCLUDE_OBJ_OPEN** and **INCLUDE_SEM_BINARY** in the project facility.

RETURNS N/A

ERRNO N/A

SEE ALSO **semOpen**

semPxLibInit( )

NAME **semPxLibInit( )** – initialize POSIX semaphore support

SYNOPSIS

```
STATUS semPxLibInit (void)
```

DESCRIPTION This routine must be called before using POSIX semaphores. If POSIX semaphores are included, this routine will be called during system initialization.

RETURNS **OK**, or **ERROR** if there is an error installing the semaphore library.

ERRNO None

SEE ALSO **semPxLib**

semPxShow( )

NAME **semPxShow( )** – display semaphore internals

SYNOPSIS

```
STATUS semPxShow
    (
    sem_t * semDesc,
    int     level
    )
```

DESCRIPTION This routine displays POSIX semaphore information. Currently, only a *level* of 0 is supported. This function prints the semaphore name, how many times sem_open has been called, and the value of the semaphore. If the semaphore value is greater than zero, than the number of available semaphores is printed. If the semaphore value is equal to 0, then the number of blocked tasks are also printed.

RETURNS **OK** or **ERROR** if the descriptor is invalid.

ERRNO N/A

SEE ALSO **semPxShow**

semPxShowInit( )

NAME **semPxShowInit( )** – initialize the POSIX semaphore show facility

SYNOPSIS

```
STATUS semPxShowInit (void)
```

DESCRIPTION This routine links the POSIX semaphore show routine into the VxWorks system. It is called automatically when the this show facility is configured into VxWorks using the **INCLUDE_POSIX_SEM_SHOW** component.

RETURNS **OK**.

ERRNO N/A

SEE ALSO **semPxShow**

semRTake( )

NAME **semRTake( )** – take a semaphore as a reader

SYNOPSIS

```
STATUS semRTake
    (
    SEM_ID semId,   /* semaphore ID to take */
    int    timeout  /* timeout in ticks */
    )
```

DESCRIPTION Takes the semaphore. If the semaphore is held by another task in "write" mode (or another task has attempted to take the semaphore in "write" mode and pended) the task will become pended until the semaphore becomes available. If the semaphore is already available or held by other tasks in "read" mode (with no tasks pended in "write" mode) the caller will gain ownership.

After a successful call to this routine the caller is granted concurrent access along with those tasks that have also taken the semaphore in this mode. Mutual exclusion is maintained between these tasks and tasks that have taken the semaphore in "write" mode.

This routine may be called recursively. However, it should not be called by a task that holds the semaphore in "write" mode. Calling **semRTake( )** in such circumstances will result in a return value of **ERROR**.

If deletion safe option is enabled, an implicit **taskSafe( )** operation will occur.

If priority inversion safe option is enabled, and the calling task blocks, and the priority of the calling task is greater than the semaphore owner, the owner will inherit the caller's priority.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

WARNING This routine must not be used from interrupt level.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the task timed out

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_objLib_OBJ_UNAVAILABLE
Would have blocked but **NO_WAIT** was specified.

S_semLib_INVALID_OPERATION
Task already holds the semaphore as a writer.

SEE ALSO **semRWLib**

semRWCreate( )

NAME **semRWCreate( )** – create and initialize a reader/writer semaphore

SYNOPSIS

```
SEM_ID semRWCreate
    (
    int options,    /* reader/writer semaphore options */
    int maxReaders  /* maximum concurrent readers */
    )
```

DESCRIPTION This routine allocates and initializes a reader/writer semaphore.

Semaphore options include the following:

SEM_Q_PRIORITY (0x1)
Queue pended tasks on the basis of their priority.

SEM_Q_FIFO (0x0)
Queue pended tasks on a first-in-first-out basis.

SEM_DELETE_SAFE (0x4)
Protect a task that owns the semaphore from unexpected deletion. This option enables an implicit **taskSafe( )** for each **semTake( )**, and an implicit **taskUnsafe( )** for each **semGive( )**.

SEM_INVERSION_SAFE (0x8)
Protect the system from priority inversion. With this option, the task or tasks owning the semaphore will execute at the highest priority of the tasks pended on the semaphore, if it is higher than its current priority. This option must be accompanied by the **SEM_Q_PRIORITY** queuing mode.

The *maxReaders* argument specifies the maximum number of tasks that may concurrently hold a read/write semaphore in **read** mode. It is an error to specify a value of **0** for *maxReaders*. If the value of *maxReaders* exceeds the system maximum value (specified in the component configuration option **SEM_RW_MAX_CONCURRENT_READERS**) then that system specific maximum will be used instead of *maxReaders*.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The semaphore ID, or **NULL** if the semaphore cannot be created.

ERRNO

S_semLib_INVALID_OPTION
Invalid option was passed to semRWCreate or *maxReaders* is 0.

S_memLib_NOT_ENOUGH_MEMORY
Not enough memory available to create the semaphore.

SEE ALSO **semLib**, **semRWLib**, **semMLib**, **semBLib**, **taskSafe( )**, **taskUnsafe( )**

semRWGiveForce( )

NAME **semRWGiveForce( )** – give a reader/writer semaphore without restrictions

SYNOPSIS

```
STATUS semRWGiveForce
    (
    FAST SEM_ID semId  /* semaphore ID to give */
    )
```

DESCRIPTION This routine gives a reader/writer semaphore, regardless of semaphore ownership. It is intended as a debugging aid only.

The routine is particularly useful when a task dies while holding some reader/writer semaphore, because the semaphore can be resurrected. The routine will give the semaphore to the next task in the pend queue or make the semaphore full if no tasks are pending. In effect, execution will continue as if the task owning the semaphore had actually given the semaphore.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

CAVEATS This routine should be used only as a debugging aid, when the condition of the semaphore is known.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid

ERRNO

S_intLib_NOT_ISR_CALLABLE
This routine is not callable from an ISR.

SEE ALSO **semRWLib**, **semGive( )**

semRWInitialize( )

NAME **semRWInitialize( )** – initialize a pre-allocated read/write semaphore.

SYNOPSIS

```
SEM_ID semRWInitialize
    (
    char * pSemMem,    /* pointer to allocated storage */
    int    options,    /* RW semaphore options */
    int    maxReaders  /* maximum concurrent readers */
    )
```

DESCRIPTION This routine initializes a reader/writer semaphore that has been pre- allocated (i.e. by the **VX_READ_WRITE_SEMAPHORE** macro). The semaphore is initialized and an ID is returned for further operations on this semaphore.

The *options* parameter has the same meaning as that for **semRWCreate( )**. Please see the documentation for **semRWCreate( )** for more details.

The *maxReaders* parameter specifies the maximum concurrent readers for the semaphores. If this value exceeds that of the system defined maximum, specified in **SEM_RW_MAX_CONCURRENT_READERS**, then that system specified value will be used instead of *maxReaders*. It is worth noting that memory allocated in this case will still be that of a semaphore created with *maxReaders* number of maximum readers. It is an error to specify **0** as the value of *maxReaders*.

The following example illustrates use of the **VX_READ_WRITE_SEMAPHORE** macro and this function together to instantiate a read/write semaphore statically (without using any dynamic memory allocation):

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

     #define NUM_READERS  0x20

     /* declare the semaphore */

     VX_READ_WRITE_SEMAPHORE(mySemRW, NUM_READERS);
     SEM_ID mySemRWId;              /* semaphore ID for further operations */

     STATUS initializeFunction (void)
         {
         if ((mySemRWId =
                     semRWInitialize (mySemRW, options, NUM_READERS)) ==
NULL)
              return (ERROR);       /* initialization failed */
         else
```

```
            return (OK);
        }
```

RETURNS The semaphore ID, or **NULL** on error

ERRNO **S_semLib_INVALID_OPTION**
Invalid options were provided.

SEE ALSO **semRWLib**

semShow( )

NAME **semShow( )** – show information about a semaphore

SYNOPSIS

```
STATUS semShow
    (
    SEM_ID semId,  /* semaphore to display */
    int    level   /* 0 = summary, 1 = details */
    )
```

DESCRIPTION This routine displays the state and optionally the pended tasks of a semaphore.

A summary of the state of the semaphore is displayed as follows:

```
    Semaphore Id        : 0x585f2
    Semaphore Type      : BINARY
    Task Queuing        : PRIORITY
    Pended Tasks        : 1
    State               : EMPTY {Count if COUNTING, Owner if MUTEX}
    Options             : 0x1       SEM_Q_PRIORITY

    VxWorks Events
    --------------
    Registered Task     : 0x594f0 (t1)
    Event(s) to Send    : 0x1
    Options             : 0x7       EVENTS_SEND_ONCE
                                    EVENTS_ALLOW_OVERWRITE
                                    EVENTS_SEND_IF_FREE
```

If *level* is 1, then more detailed information will be displayed. If tasks are blocked on the queue, they are displayed in the order in which they will unblock, as follows:

```
    Pended Tasks
    ------------
       NAME      TID    PRI DELAY
    ---------- -------- --- -----
    tExcTask    3fd678   0    21
    tLogTask    3f8ac0   0   611
```

RETURNS

OK or **ERROR**.

ERRNO

S_smObjLib_NOT_INITIALIZED
The shared memory object library is not initialized.

SEE ALSO

semShow, **windsh**, *VxWorks Programmer's Guide*, *VxWorks Command-Line Tools User's Guide*.

semTake( )

NAME

semTake( ) – take a semaphore

SYNOPSIS

```
STATUS semTake
    (
    SEM_ID semId,   /* semaphore ID to take */
    int    timeout  /* timeout in ticks */
    )
```

DESCRIPTION

This routine performs the take operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and the calling task may be affected. The behavior of **semTake( )** is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, **semTake( )** will return **ERROR**. Timeouts of **WAIT_FOREVER** (-1) and **NO_WAIT** (0) indicate to wait indefinitely or not to wait at all.

When **semTake( )** returns due to timeout, it sets the errno to **S_objLib_OBJ_TIMEOUT** (defined in **objLib.h**).

The **semTake( )** routine is not callable from interrupt service routines.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_objLib_OBJ_UNAVAILABLE
Would have blocked but **NO_WAIT** was specified.

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semRWLib**, **semSmLib**

semUnlink( )

NAME **semUnlink( )** – unlink a named semaphore

SYNOPSIS

```
STATUS semUnlink
    (
    const char * name  /* name of semaphore to unlink */
    )
```

DESCRIPTION This routine removes a semaphore from the name space, and marks it as ready for deletion on the last **semClose( )**. In case there are already no outstanding **semOpen( )** calls, the semaphore is deleted. After a semaphore is unlinked, subsequent calls to **semOpen( )** using *name* will not be able to find the semaphore, even if it has not been deleted yet. Instead, a new semaphore could be created if **semOpen( )** is called with the **OM_CREATE** flag.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_objLib_OBJ_INVALID_ARGUMENT**
name is **NULL** or empty.

S_objLib_OBJ_NOT_FOUND
No semaphore with *name* was found.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Semaphore is not named.

S_objLib_OBJ_DESTROY_ERROR
Error while deleting the semaphore.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **semOpen**, **semOpen( )**, **semClose( )**

semWTake( )

NAME **semWTake( )** – take a semaphore in write mode

SYNOPSIS

```
STATUS semWTake
    (
    SEM_ID semId,   /* semaphore ID to take */
    int    timeout  /* timeout in ticks */
    )
```

DESCRIPTION Takes the semaphore. If the semaphore is not available, i.e., it is held in either "read" or "write" mode by another task, this task will become pended until the semaphore becomes available. If the semaphore is already available this call will take the semaphore and continue running.

After a successful call to this routine the caller is granted exclusive access to the resource.

This routine may be called recursively. However, it should not be called by a task that holds the semaphore in "read" mode. Calling **semWTake( )** in such circumstances will result in a return value of **ERROR**.

If deletion safe option is enabled, an implicit **taskSafe( )** operation will occur.

If priority inversion safe option is enabled, and the calling task blocks, and the priority of the calling task is greater than the semaphore owner, the owner will inherit the caller's priority.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

WARNING This routine must not be used from interrupt level.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the task timed out

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_objLib_OBJ_UNAVAILABLE
Would have blocked but **NO_WAIT** was specified.

S_semLib_INVALID_OPERATION
Task already holds the semaphore as a writer.

SEE ALSO **semRWLib**

sem_close( )

NAME **sem_close( )** – close a named semaphore (POSIX)

SYNOPSIS

```
int sem_close
    (
    sem_t * sem  /* semaphore descriptor */
    )
```

DESCRIPTION This routine is called to indicate that the calling task is finished with the specified named semaphore, *sem*. It deallocates any system resources allocated by the system for use by this task for this semaphore. Calling **sem_close( )** with an unnamed semaphore will result in an **EINVAL** error.

If the semaphore has not been removed with a call to **sem_unlink( )**, then **sem_close( )** has no effect on the state of the semaphore. However, if the semaphore has been unlinked, it is destroyed when the last reference to it is closed.

WARNING Take care to avoid risking the deletion of a semaphore that another task has already locked. Applications should only close semaphores that the closing task has opened.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO **EINVAL** – invalid semaphore descriptor – the semaphore is unnamed

SEE ALSO **semPxLib**, **sem_unlink( )**, **sem_open( )**, **sem_init( )**

sem_destroy( )

NAME

sem_destroy( ) – destroy an unnamed semaphore (POSIX)

SYNOPSIS

```
int sem_destroy
    (
    sem_t * sem  /* semaphore descriptor */
    )
```

DESCRIPTION

This routine is used to destroy the unnamed semaphore indicated by *sem*.

The **sem_destroy( )** call can only destroy a semaphore created by **sem_init( )**. Calling **sem_destroy( )** with a named semaphore will cause an **EINVAL** error. Subsequent use of the *sem* semaphore after destruction will cause an **EINVAL** error.

If one or more tasks is blocked on the semaphore, the semaphore is not destroyed, and the routine returns with **EBUSY** error.

WARNING

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.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL – invalid semaphore descriptor – the specified semaphore, *sem*, is named
EBUSY – one or more tasks is blocked on the semaphore

SEE ALSO

semPxLib, **sem_init( )**

sem_getvalue( )

NAME

sem_getvalue( ) – get the value of a semaphore (POSIX)

SYNOPSIS

```
int sem_getvalue
    (
    sem_t * sem,  /* semaphore descriptor */
    int *   sval  /* buffer by which the value is returned */
    )
```

DESCRIPTION

This routine updates the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but may not be the actual value of the semaphore when it is returned to the calling task.

If *sem* is locked, the value returned by **sem_getvalue( )** will either be zero or a negative number whose absolute value represents the number of tasks waiting for the semaphore at some unspecified time during the call.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL – invalid semaphore descriptor – invalid *sval* pointer

SEE ALSO

semPxLib, **sem_post( )**, **sem_trywait( )**, **sem_trywait( )**

sem_init( )

NAME

sem_init( ) – initialize an unnamed semaphore (POSIX)

SYNOPSIS

```
int sem_init
    (
    sem_t *      sem,      /* semaphore to be initialized */
    int          pshared,  /* RTP sharing :ignored */
    unsigned int value     /* semaphore initialization value */
    )
```

DESCRIPTION

This routine is used to initialize the unnamed semaphore *sem*. The value of the initialized semaphore is *value*. Following a successful call to **sem_init( )** the semaphore may be used in subsequent calls to **sem_wait( )**, **sem_trywait( )**, and **sem_post( )**. This semaphore remains usable until the semaphore is destroyed.

The value of *pshared* parameter is ignored.

Only *sem* itself maybe used for performing synchronization. The result of referring to copies of *sem* in calls to sem_wait, **sem_trywait( )**, **sem_post( )** and **sem_destroy( )** is undefined.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO
EINVAL – *value* exceeds
SEM_VALUE_MAX – *sem* points to an invalid buffer
ENOSPC – unable to initialize semaphore due to resource constraints

SEE ALSO **semPxLib**, **sem_wait( )**, **sem_trywait( )**, **sem_post( )**, **sem_destroy( )**

sem_open( )

NAME **sem_open( )** – initialize/open a named semaphore (POSIX)

SYNOPSIS

```
sem_t * sem_open
    (
    const char * name,   /* semaphore name */
    int          oflag,  /* semaphore creation flags */
    ...                  /* extra optional parameters */
    )
```

DESCRIPTION This routine establishes a connection between a named semaphore and a task. Following a call to **sem_open( )** with a semaphore name *name*, the task may reference the semaphore associated with *name* using the address returned by this call. This semaphore may be used in subsequent calls to **sem_wait( )**, **sem_trywait( )**, and **sem_post( )**. The semaphore remains usable until the semaphore is closed by a successful call to **sem_close( )**.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to **sem_open( )**. The following flag bits may be set in *oflag*:

O_CREAT
Use this flag to create a semaphore if it does not already exist. If **O_CREAT** is set and the semaphore already exists, **O_CREAT** has no effect except as noted below under **O_EXCL**. Otherwise, **sem_open( )** creats a semaphore. **O_CREAT** requires a third and fourth argument: *mode*, which is of type mode_t, and *value*, which is of type unsigned int. *mode* has no effect in this implementation. The semaphore is created with an initial value of *value*. Valid initial values for semaphores must be less than or equal to **SEM_VALUE_MAX**.

O_EXCL
If **O_EXCL** and **O_CREAT** are set, **sem_open( )** will fail if the semaphore name exists. If **O_EXCL** is set and **O_CREAT** is not set, the named semaphore is not created.

To determine whether a named semaphore already exists in the system, call **sem_open( )** with the flags **O_CREAT | O_EXCL**. If the **sem_open( )** call fails, the semaphore exists.

The semaphore must have a name. **NULL** or empty strings result in **EINVAL**. If the semaphore *name* begins with the slash character, then it is treated as a public semaphore.

RTPs can open their own references to the public semaphore by using its name. If the *name* does not begin with the slash character, then it is treated as a private semaphore and RTPs cannot get access to it.

If a task makes multiple calls to **sem_open( )** with the same value for *name*, then a reference to the same semaphore is returned for each such call, provided that there have been no calls to **sem_unlink( )** for this semaphore.

References to copies of the semaphore will produce undefined results.

NOTE
The current implementation has the following limitations:

- A semaphore cannot be closed with calls to **_exit( )** or **exec( )**.
- A semaphore cannot be implemented as a file.
- Semaphore names will not appear in the file system.

RETURNS
A pointer to sem_t, or -1 (**ERROR**) if unsuccessful.

ERRNO
EEXIST -
O_CREAT and
O_EXCL are set and the semaphore already exists
EINVAL – *value* exceeds
SEM_VALUE_MAX – the semaphore name is invalid
ENAMETOOLONG – the semaphore name is too long
ENOENT – the named semaphore does not exist and
O_CREAT is not set
ENOSPC – the semaphore could not be initialized due to resource constraints

SEE ALSO
semPxLib, **sem_unlink( )**, **sem_close( )**

sem_post( )

NAME
sem_post( ) – unlock (give) a semaphore (POSIX)

SYNOPSIS
```
int sem_post
    (
    sem_t * sem  /* semaphore descriptor */
    )
```

DESCRIPTION
This routine unlocks the semaphore referenced by *sem* by performing the semaphore unlock operation on that semaphore.

If the semaphore value resulting from the operation is positive, then no tasks were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this semaphore is zero, then one of the tasks blocked waiting for the semaphore will return successfully from its call to **sem_wait( )**.

NOTE

The **_POSIX_PRIORITY_SCHEDULING** functionality is not yet supported.

Note that the POSIX terms *unlock* and *post* correspond to the term *give* used in other VxWorks semaphore documentation.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL – invalid semaphore descriptor

SEE ALSO

semPxLib, **sem_wait( )**, **sem_trywait( )**

sem_timedwait( )

NAME

sem_timedwait( ) – lock (take) a semaphore with a timeout (POSIX)

SYNOPSIS

```
int sem_timedwait
    (
    sem_t *                  sem,
    const struct timespec * abs_timeout
    )
```

DESCRIPTION

This routine locks the semaphore referenced by *sem*. If the semaphore cannot be locked immediately, the calling process will wait till the absolute time specified by *abs_timeout* passes. If the semaphore cannot be locked before *abs_timeout* has passed, an error is returned.

Upon successful return, the state of the semaphore is always locked (either as a result of this call or by a previous **sem_wait( )** or **sem_trywait( )**). The semaphore will remain locked until **sem_post( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks semaphore documentation.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

ETIMEDOUT
The semaphore could not be locked before the timeout expired.

EINVAL
The semaphore descriptor is invalid, or the nanosecond field of the timeout value is greater than 1 billion.

EINTR
A signal interrupted this function.

SEE ALSO

semPxLib, **sem_wait( )**, **sem_trywait( )**, **sem_post( )**

sem_trywait( )

NAME

sem_trywait( ) – lock (take) a semaphore, returning error if unavailable (POSIX)

SYNOPSIS

```
int sem_trywait
    (
    sem_t * sem  /* semaphore descriptor */
    )
```

DESCRIPTION

This routine locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore. In either case, this call returns immediately without blocking.

Upon successful return, the state of the semaphore is always locked (either as a result of this call or by a previous **sem_wait( )** or **sem_trywait( )**). The semaphore will remain locked until **sem_post( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks semaphore documentation.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO **EAGAIN** – semaphore is already locked
EINVAL – invalid semaphore descriptor

SEE ALSO **semPxLib**, **sem_wait( )**, **sem_post( )**

sem_unlink( )

NAME **sem_unlink( )** – remove a named semaphore (POSIX)

SYNOPSIS

```
int sem_unlink
    (
    const char * name  /* semaphore name */
    )
```

DESCRIPTION This routine removes the string *name* from the semaphore name table, and marks the corresponding semaphore for destruction. An unlinked semaphore is destroyed when the last reference to it is removed by **sem_close( )**. After a name is unlinked, calls to **sem_open( )** using the same name cannot connect to the same semaphore, even if other tasks are still using it. Instead, such calls refer to a new semaphore with the same name.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO **ENAMETOOLONG** – semaphore name too long
ENOENT – a semaphore with the specified *name* does not exist

SEE ALSO **semPxLib**, **sem_open( )**, **sem_close( )**

sem_wait( )

NAME

sem_wait( ) – lock (take) a semaphore, blocking if not available (POSIX)

SYNOPSIS

```
int sem_wait
    (
    sem_t * sem  /* semaphore descriptor */
    )
```

DESCRIPTION

This routine locks the semaphore referenced by *sem* by performing the semaphore lock operation on that semaphore. If the semaphore value is currently zero, the calling task will not return from the call to **sem_wait( )** until it either locks the semaphore or the call is interrupted by a signal.

On return, the state of the semaphore is locked and will remain locked until **sem_post( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks documentation regarding semaphores.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL – invalid semaphore descriptor
EINTR – got a signal while blocking on the semaphore. Applicable only for RTP task

SEE ALSO

semPxLib, **sem_trywait( )**, **sem_post( )**

set_new_handler( )

NAME

set_new_handler( ) – set new_handler to user-defined function (C++)

SYNOPSIS

```
extern void (*set_new_handler (void(* pNewNewHandler)())) (void)
```

DESCRIPTION

This function is used to define the function that will be called when operator new cannot allocate memory.

The new_handler acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS A pointer to the previous value of new_handler.

ERRNO Not Available

SEE ALSO **cplusLib**

set_terminate( )

NAME **set_terminate( )** – set terminate to user-defined function (C++)

SYNOPSIS

```
extern void (*set_terminate (void(* terminate_handler)())) (void)
```

DESCRIPTION This function is used to define the terminate_handler which will be called when an uncaught exception is raised.

The terminate_handler acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS The previous terminate_handler.

ERRNO Not Available

SEE ALSO **cplusLib**

shConfig( )

NAME **shConfig( )** – display or set the shell configuration

SYNOPSIS

```
void shConfig
    (
    const char * config  /* configuration string */
    )
```

DESCRIPTION This routine displays or sets the shell configuration of the current shell session.

If *config* is **NULL**, the routine displays the cofiguration variables; otherwise, it sets the configuration. The format of the string *config* is:

```
<variable> = <value> , <variable> = <value> , ...
```

The variable name or value can contain **,** and **=** if these characters are escaped or quoted.

RETURNS N/A.

ERRNO N/A

SEE ALSO **usrLib**, **shellConfigGet( )**, **shellConfigSet( )**, the VxWorks programmer guides.

shellAbort( )

NAME **shellAbort( )** – abort a shell session

SYNOPSIS

```
STATUS shellAbort
    (
    SHELL_ID shellId  /* shell session Id */
    )
```

DESCRIPTION This routine aborts the shell session *shellId*. Before the shell task is restarted, its task trace is printed.

If *shellId* is equal to **ALL_SHELL_SESSIONS**, all the shell session are aborted. if *shellId* is equal to **CURRENT_SHELL_SESSION**, the current shell session is aborted.

RETURNS **OK**, or **ERROR** if *shellId* is not a valid shell session Id.

ERRNO N/A

SEE ALSO **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellCmdAdd( )

NAME **shellCmdAdd( )** – add a shell command

SYNOPSIS

```
STATUS shellCmdAdd
    (
    const char *      topic,     /* topic name */
    const SHELL_CMD * pShellCmd  /* pointer on the command structure */
    )
```

DESCRIPTION

This routine adds the shell command pointed by *pShellCmd*. The command is added under the topic *topic*. If the topic does not already exist, an error is returned.

COMMAND FORMAT

A command is defined as a structure: {*cmdFullname*, *func*, *opt*, *shortDesc*, *fullDesc*, *synopsis*}

cmdFullname is the name of the command to add. It may be a composed command name, that means a command name like "foo bar". In that example, "foo" is the top command name, and "bar" is a sub-command of "foo".

func is the function to call for that command name.

If the *opt* parameter is not equal to **NULL**, the declaration of that function is:

```
int func
    (
    SHELL_OPTION options[]   /* options array */
    )
```

options is a pointer on the argument array of the command.

If the *opt* parameter is **NULL**, the declaration of the function is:

```
int func
    (
    int      argc,           /* number of argument */
    char **   arcv           /* pointer on the array of arguments */
    )
```

opt is a string that describes the possible options that the command accepts. Each option is a single character (case sensitive) which defines a single option. If an option expects an extra argument, an extra **:** character has to be added after the option character. For example, "avf:" means that the command accepts options "-a", "-v" and "-f *extraArg*". The order of the option character matters: it defines the order of the options in the option array passed to *func*.

Each cell of the options array passed to *func* is composed of a boolean value (**TRUE** if the option is set, **FALSE** otherwise) and of a pointer on a string (pointer on an extra argument). Another boolean indicates if it is the last cell of the array.

If the option string *opt* is ":", the argument string of the command is passed to *func* without any process, as the string field of the first cell of the options array.

shortDesc is a short description of the command. A sequence of "%s" characters will be replaced by the function name. The string should not be ended by a \n character.

fullDesc is the full description of the command. A sequence of "%s" characters will be replaced by the function name. This description should contain the explanation of the command options. The string should not be ended by a \n character.

synopsis is the synopsis of the command. A sequence of "%s" characters will be replaced by the function name. The string should not be ended by a \n character.

RETURNS

OK, or **ERROR** if an error occured

ERRNO

S_shellInterpCmdLib_UNKNOWN_TOPIC
The topic is not registered yet.

S_shellLib_UNMATCHED_QUOTE
A quote character is missing in the command name

S_shellInterpCmdLib_WRONG_CMD
A wrong command name was supplied

malloc( ), **calloc( )** and **memPartCreate( )** errnos.

SEE ALSO

shellInterpCmdLib, **shellCmdTopicAdd( )**, **shellCmdArrayAdd( )**.

shellCmdAliasAdd( )

NAME

shellCmdAliasAdd( ) – add an alias string

SYNOPSIS

```
STATUS shellCmdAliasAdd
    (
    const char * aliasName,  /* alias name */
    const char * string,     /* string aliased */
    BOOL         allocate    /* TRUE to allocate memory */
    )
```

DESCRIPTION

This routine adds the alias string *aliasName*. During the parsing of the shell imput line by the command interpreter, this *aliasName* string will be replaced by *string*.

If *allocate* is set to **FALSE**, both strings are not copied into an internal database; only their pointers are stored for future use. If *allocate* is **TRUE**, a buffer is allocated to store the strings *aliasName* and *string*. This buffer is freed when the alias is removed.

NOTE

It is not possible to add an alias if a command with the same name exists.

RETURNS

OK, or **ERROR** if the alias cannot be added.

ERRNO

N/A

SEE ALSO

shellInterpCmdLib, **shellCmdAliasArrayAdd( )**, **shellCmdAliasDelete( )**.

shellCmdAliasArrayAdd( )

NAME **shellCmdAliasArrayAdd( )** – add an array of alias strings

SYNOPSIS

```
STATUS shellCmdAliasArrayAdd
    (
    const SHELL_CMD_ALIAS aliasArray[],  /* array of aliases */
    BOOL                  allocate       /* TRUE to allocate memory */
    )
```

DESCRIPTION This routine adds the alias stored in the array *aliasArray*. The end of the array is defined by an entry with the alias name set to **NULL**.

If *allocate* is set to **FALSE**, both strings are not copied into an internal database; only their pointers are stored for future use. If *allocate* is **TRUE**, a buffer is allocated to store the strings *aliasName* and *string*. This buffer is freed when the alias is removed.

NOTE It is not possible to add an alias if a command with the same name exists.

RETURNS **OK**, or **ERROR** if one alias cannot be added.

ERRNO N/A

SEE ALSO **shellInterpCmdLib**, **shellCmdAliasAdd( )**

shellCmdAliasDelete( )

NAME **shellCmdAliasDelete( )** – delete an alias

SYNOPSIS

```
STATUS shellCmdAliasDelete
    (
    const char * alias  /* alias to delete */
    )
```

DESCRIPTION This routine deletes the alias *alias*.

RETURNS **OK**, or **ERROR** if the alias cannot be found.

ERRNO N/A

SEE ALSO **shellInterpCmdLib**, **shellCmdAliasAdd( )**

shellCmdArrayAdd( )

NAME **shellCmdArrayAdd( )** – add an array of shell commands

SYNOPSIS

```
STATUS shellCmdArrayAdd
    (
    const char *    topic,            /* topic name */
    const SHELL_CMD shellCmdArray[]  /* array of commands */
    )
```

DESCRIPTION This routine adds the list of shell commands stored in the array pointed by *pShellCmdArray*. These commands are added under the topic *topic*. If the topic does not already exist, an error is returned.

COMMAND ARRAY FORMAT

An element of the array is a command structure as described by the routine **shellCmdAdd( )**. The end of the array is marked by a *cmdFullname* equals to **NULL**.

RETURNS **OK**, or **ERROR** if one command cannot be added.

ERRNO

S_shellInterpCmdLib_UNKNOWN_TOPIC
The topic is not registered yet.

S_shellLib_UNMATCHED_QUOTE
A quote character is missing in the command name

S_shellInterpCmdLib_WRONG_CMD
A wrong command name was supplied

malloc( ), **calloc( )** and **memPartCreate( )** errno.

SEE ALSO **shellInterpCmdLib**, **shellCmdTopicAdd( )**, **shellCmdAdd( )**.

shellCmdExec( )

NAME **shellCmdExec( )** – execute a shell command

SYNOPSIS

```
STATUS shellCmdExec
    (
    const char * name,  /* command name to execute */
    const char * args   /* arguments of the command */
    )
```

DESCRIPTION

This routine executes the shell command *name* with the argument string *args*. The argument string *args* is parsed to extract all arguments and options.

The arguments are separated by blank characters. If a blank character has to be set as part of an argument, the argument needs to be quoted (with simple or double quote) or the space has to be escaped (with the "\" character). The argument string is parsed using the option string set for the command.

If the pointer to the option string of the command is **NULL**, the arguments are only split into an array of strings. This array and the number of strings is passed to the command function (as argc/argv parameters).

This routine has to be called from within a shell task. It cannot be called by another task.

RETURNS

OK, or **ERROR** if the command cannot be executed.

ERRNO

S_shellInterpCmdLib_UNKNOWN_CMD
The command name *name* is unknown.

S_shellLib_UNKNOWN_OPT
A options of the command is not valid.

S_shellLib_UNMATCHED_QUOTE
A quote string is not ended with a quote character.

S_shellLib_MISSING_ARG
An option is missing its extra argument.

malloc( ) and **memPartCreate( )** errno.

SEE ALSO

shellInterpCmdLib

shellCmdMemRegister( )

NAME

shellCmdMemRegister( ) – register a buffer against the command interpreter

SYNOPSIS

```
STATUS shellCmdMemRegister
    (
    void * pMem,      /* memory block to register */
    BOOL   shellPool  /* TRUE for shell memory pool */
    )
```

DESCRIPTION

This routine registers the memory block pointed by *pMem* into the memory list of the command interpreter. When the shell task is restarted or ended, the registered memory blocks are freed by the shell.

If the memory block has been allocated from the shell memory pool using **shellMemMalloc( )** or **shellMemCalloc( )** routines, *shellPool* must be set to to **TRUE**. Otherwise, *shellPool* must be set to **FALSE**.

RETURNS **OK**, or **ERROR** if an error occured.

ERRNO **S_shellLib_NO_SHELL_CMD**
The current interpreter of the current shell session is not the command interpreter.

malloc( ) and **memPartCreate( )** errno.

SEE ALSO **shellInterpCmdLib**, **shellCmdMemRegister( )**

shellCmdMemUnregister( )

NAME **shellCmdMemUnregister( )** – unregister a buffer

SYNOPSIS

```
STATUS shellCmdMemUnregister
    (
    void * pMem  /* memory block to unregister */
    )
```

DESCRIPTION This routine unregisters the memory block pointed by *pMem* from the memory list of the command interpreter.

RETURNS **OK**, or **ERROR** if the memory was not registered

ERRNO **S_shellLib_NO_SHELL_CMD**
The current interpreter of the current shell session is not the command interpreter.

SEE ALSO **shellInterpCmdLib**, **shellCmdMemRegister( )**.

shellCmdPreParseAdd( )

NAME **shellCmdPreParseAdd( )** – define a command to be pre-parsed

SYNOPSIS

```
STATUS shellCmdPreParseAdd
    (
    const char *              name,  /* command name to pre-parse */
    char * (*preParseRtn) (char * line)  /* pre-parse routine address */
    )
```

DESCRIPTION

This routine is used to define that the command *name* needs a special handling. During the command line parsing, if *name* string is found at the beginning of a line, the routine *preParseRtn* is called before any processing by the interpreter. *name* must be a single command name.

The prototype of *preParseRtn* is:

```
char * preParseRtn
    {
    char * line       /* complete line to be parsed */
    };
```

The *line* string is the line parsed by the interpreter, the initial blank characters being stripped. This string begins with the command *name*.

If *preParseRtn* returns **NULL**, the interpreter interupts the line parsing. Otherwise, it continues the parsing of the returned pre-parsed line.

The returned buffer must be allocated by the routine *preParseRtn* using either a **malloc( )**, **calloc( )** or **strdup( )** call. It will be freed by the interpreter using **free( )**.

RETURNS

OK, or **ERROR** if the command *name* is not registered yet

ERRNO

N/A

SEE ALSO

shellInterpCmdLib

shellCmdSymTabIdGet( )

NAME

shellCmdSymTabIdGet( ) – get symbol table Id of a shell session

SYNOPSIS

```
SYMTAB_ID shellCmdSymTabIdGet
    (
    SHELL_ID shellId
    )
```

DESCRIPTION

This routine returns the symbol table Id associated to the current memory context of the shell session *shellId*. *shellId* can be equal to **CURRENT_SHELL_SESSION**.

RETURNS

the symbol table Id, or 0 if the symbol table is not available

ERRNO

N/A

SEE ALSO **shellInterpCmdLib**

shellCmdTopicAdd( )

NAME **shellCmdTopicAdd( )** – add a shell command topic

SYNOPSIS

```
STATUS shellCmdTopicAdd
    (
    const char * topic,  /* topic name */
    const char * desc    /* topic description */
    )
```

DESCRIPTION This routine adds the topic *topic* with the description *desc* to the list of topics.

NOTE The topic name and the description are not copied internaly. Only the string pointers are stored in the internal structure.

RETURNS **OK**, or **ERROR** if the topic cannot be added.

ERRNO N/A

SEE ALSO **shellInterpCmdLib**

shellCompatibleCheck( )

NAME **shellCompatibleCheck( )** – check the compatibility mode of the shell

SYNOPSIS

```
BOOL shellCompatibleCheck (void)
```

DESCRIPTION This routine checks if the shell is configured to be VxWorks 5.5 compatible or not.

RETURNS **TRUE** if the shell is VxWorks 5.5 compatible, **FALSE** otherwise.

ERRNO N/A

SEE ALSO **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigDefaultGet( )

NAME
shellConfigDefaultGet( ) – get default shell configuration

SYNOPSIS
```
char * shellConfigDefaultGet (void)
```

DESCRIPTION
This routine returns the default shell configuration variables. The format of the string returned is:

```
<variable>=<value>,<variable>=<value>, ...
```

The returned string has to be freed by the caller with a **free( )**.

RETURNS
a pointer on the default shell configuration string, or **NULL** if an error occured.

ERRNO
N/A

SEE ALSO
shellConfigLib, **shellConfigGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigDefaultSet( )

NAME
shellConfigDefaultSet( ) – set default shell configuration

SYNOPSIS
```
STATUS shellConfigDefaultSet
    (
    const char * config  /* default configuration string */
    )
```

DESCRIPTION
This routine sets the default values of the shell configuration variables. The format of the string *config* is:

```
<variable> = <value> , <variable> = <value> , ...
```

The variable name or value can contain "," and "=" characters if they are escaped or quoted.

These configuration variables are seen by all shell sessions and can be superseed by a configuration variable defined localy to a shell session.

RETURNS
OK, **ERROR** if the configuration cannot be set or there was a problem

ERRNO
S_shellLib_CONFIG_ERROR
    The format of the configuration string is wrong.

malloc( ) and **memPartCreate( )** errno.

SEE ALSO **shellConfigLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigDefaultValueGet( )

NAME **shellConfigDefaultValueGet( )** – get a default configuration variable value

SYNOPSIS

```
const char * shellConfigDefaultValueGet
    (
    const char * name  /* variable name */
    )
```

DESCRIPTION This routine gets the value of the default configuration variable *name*. If the variable does not exist, **NULL** is returned.

RETURNS a pointer on the string value, or **NULL** if the variable does not exist or if there was a problem.

ERRNO N/A

SEE ALSO **shellConfigLib**, **shellConfigValueGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigDefaultValueSet( )

NAME **shellConfigDefaultValueSet( )** – set a default configuration variable value

SYNOPSIS

```
STATUS shellConfigDefaultValueSet
    (
    const char * name,  /* variable name */
    const char * value  /* variable value or NULL */
    )
```

DESCRIPTION This routine sets the value of the default configuration variable named *name*. If the variable does not exist, it is added to the default configuration variable list.

The strings *name* and *values* are copied into memory. *value* can be a **NULL** pointer.

RETURNS **OK**, **ERROR** if the configuration cannot be set or there was a problem

ERRNO N/A

SEE ALSO **shellConfigLib**, **shellConfigValueSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigDefaultValueUnset( )

NAME

shellConfigDefaultValueUnset( ) – unset a default configuration variable value

SYNOPSIS

```
void shellConfigDefaultValueUnset
    (
    const char * name  /* variable name */
    )
```

DESCRIPTION

This routine unsets the default configuration variable *name*.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

shellConfigLib, **shellConfigValueUnset( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigGet( )

NAME

shellConfigGet( ) – get the shell configuration

SYNOPSIS

```
char * shellConfigGet
    (
    SHELL_ID shellId  /* shell session Id */
    )
```

DESCRIPTION

This routine returns the shell configuration variables of the shell session *shellId*. The format of the string returned is:

```
<variable>=<value>,<variable>=<value>, ...
```

shellId can be **CURRENT_SHELL_SESSION**.

The returned string has to be freed by the caller with a **free( )**.

RETURNS

a pointer on the shell configuration string, or **NULL** if an error occured.

ERRNO

S_shellLib_NOT_SHELL_TASK
The shell session is invalid.

shellDataAdd( ), **shellDataGet( )**, **malloc( )** and **memPartCreate( )** errno.

SEE ALSO **shellConfigLib**, **shellConfigDefaultGet( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigSet( )

NAME **shellConfigSet( )** – set shell configuration

SYNOPSIS

```
STATUS shellConfigSet
    (
    SHELL_ID     shellId,  /* shell session identifier */
    const char * config    /* default configuration string */
    )
```

DESCRIPTION This routine sets the values of the shell configuration variables of the shell session *shellId*. The format of the string *config* is:

```
<variable> = <value> , <variable> = <value> , ...
```

The variable name or value can contain "," and "=" characters if they are escaped or quoted.

The configuration variables will only be seen by the shell session *shellId*, They superseed any default configuration values with same name.

shellId can be **CURRENT_SHELL_SESSION**.

RETURNS **OK**, **ERROR** if the configuration cannot be set or if *shellId* is not a valid shell session identifier.

ERRNO

S_shellLib_CONFIG_ERROR
The format of the configuration string is wrong.

S_shellLib_NOT_SHELL_TASK
The shell session is invalid.

shellDataAdd( ), **shellDataGet( )**, **malloc( )** and **memPartCreate( )** errno.

SEE ALSO **shellConfigLib**, **shellConfigDefaultSet( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigValueGet( )

NAME **shellConfigValueGet( )** – get a shell configuration variable value

SYNOPSIS

```
const char * shellConfigValueGet
```

```
    (
    SHELL_ID     shellId,  /* shell session Id */
    const char * name      /* variable name */
    )
```

DESCRIPTION This routine gets, from the shell session *shellId*, the value of the configuration variable *name*. If the variable does not exist or if *shellId* is not a valid shell session, **NULL** is returned.

shellId can be **CURRENT_SHELL_SESSION**.

RETURNS a pointer on the string value, or **NULL** if the variable does not exist or an error occurred.

ERRNO N/A

SEE ALSO **shellConfigLib**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigValueSet( )

NAME **shellConfigValueSet( )** – set a shell configuration variable value

SYNOPSIS

```
STATUS shellConfigValueSet
    (
    SHELL_ID     shellId,  /* shell session Id */
    const char * name,     /* variable name */
    const char * value     /* variable value */
    )
```

DESCRIPTION This routine sets, for the shell session *shellId*, the value of the configuration variable *name* to *value*. If the variable does not exist, it is added to the internal list of the shell session.

The strings *name* and *values* are copied into memory. *value* can be a **NULL** pointer.

shellId can be **CURRENT_SHELL_SESSION**.

RETURNS **OK**, **ERROR** if the configuration cannot be set

ERRNO **S_shellLib_NOT_SHELL_TASK**
The shell session is invalid.

shellDataAdd( ), **shellDataGet( )**, **malloc( )** and **memPartCreate( )** errno.

SEE ALSO **shellConfigLib**, **shellConfigDefaultValueSet( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellConfigValueUnset( )

NAME

shellConfigValueUnset( ) – unset a shell configuration variable value

SYNOPSIS

```
STATUS shellConfigValueUnset
    (
    SHELL_ID     shellId,  /* shell session Id */
    const char * name      /* variable name */
    )
```

DESCRIPTION

This routine unsets, for the shell session *shellId*, the configuration variable *name*.

shellId can be **CURRENT_SHELL_SESSION**.

RETURNS

OK, or **ERROR** if the variable cannot be unset.

ERRNO

S_shellLib_NOT_SHELL_TASK
The shell session is invalid.

shellDataAdd( ), **shellDataGet( )**, **malloc( )** and **memPartCreate( )** errno.

SEE ALSO

shellConfigLib, **shellConfigDefaultValueUnset( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataAdd( )

NAME

shellDataAdd( ) – add user data to a specified shell

SYNOPSIS

```
STATUS shellDataAdd
    (
    SHELL_ID           shellId,     /* the shell identifier */
    const char *       key,         /* the key for the value to add */
    void *             pData,       /* the value to add */
    SHELL_DATA_FUNCPTR finalizeRtn  /* finalize routine of the data, or NULL
*/
    )
```

DESCRIPTION

This routine adds to the user data list of the shell *shellId* a new data *pData* with the key named *key*. If the key already exists, the previous data is overwritten.

The routine *finalizeRtn* is called when the shell session is terminated to free any internal value associated with the user data *pData*. *finalizeRtn* may be **NULL**.

The prototype of *finalizeRtn* is **SHELL_DATA_FUNCPTR**:

```
void finalizeRtn
```

```
    (
    SHELL_ID      shellId,    /* shell session Id */
    const char *  key,        /* data key */
    void *        pData       /* data value */
    )
```

with *shellId* the identifier of the shell session terminated and *pData* the user data associated with the key *key*.

RETURNS **OK**, or **ERROR** if an error occured

ERRNO **S_shellLib_NOT_SHELL_TASK**
shellId is not a valid shell session.

malloc( ) and **memPartCreate( )** errno.

SEE ALSO **shellDataLib**, **shellFromNameDataAdd( )**, **shellDataRemove( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataFirst( )

NAME **shellDataFirst( )** – get the first user data that matchs a key

SYNOPSIS

```
SHELL_ID shellDataFirst
    (
    const char * key,    /* key data to search */
    void **      ppData  /* where to store data value */
    )
```

DESCRIPTION This routine returns the first user data named *key* in all shell sessions. The user data value is stored into the location pointed by *ppData*.

RETURNS the shell session Id, or 0 if none shell session contains the key *key*.

ERRNO N/A

SEE ALSO **shellDataLib**, **shellDataNext( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataFromNameAdd( )

NAME

shellDataFromNameAdd( ) – add user data to a specified shell

SYNOPSIS

```
STATUS shellDataFromNameAdd
    (
    const char *        taskName,    /* the shell task name */
    const char *        key,         /* the key for the value to add */
    void *              pData,       /* the value to add */
    SHELL_DATA_FUNCPTR finalizeRtn  /* finalize routine of the data, or NULL
*/
    )
```

DESCRIPTION

This routine adds to the user data list of shell session associated to the shell task named *taskName* a new data *pData* with the key named *key*. If the key already exists, the previous data is overwritten.

The routine *finalizeRtn* is called when the shell session is terminated to free any internal value associated with the user data *pData*. *finalizeRtn* may be **NULL**.

The prototype of *finalizeRtn* is **SHELL_DATA_FUNCPTR**:

```
void finalizeRtn
    (
    SHELL_ID      shellId,    /* shell session Id */
    const char *  key,        /* data key */
    void *        pData       /* data value */
    )
```

with *shellId* the identifier of the shell session terminated and *pData* the user data associated with the key *key*.

RETURNS

OK, or **ERROR** if an error occured.

ERRNO

S_shellLib_NOT_SHELL_TASK
taskName is not a shell task.

malloc( ) and **memPartCreate( )** errno.

SEE ALSO

shellDataLib, **shellDataAdd( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataFromNameGet( )

NAME

shellDataFromNameGet( ) – get user data from a specified shell

SYNOPSIS

```
STATUS shellDataFromNameGet
```

```
    (
    const char * taskName,  /* the shell task name */
    const char * key,       /* the key for the value to get */
    void **      ppData     /* get the data there */
    )
```

DESCRIPTION This routine returns the data named *key* stored within the shell session whose associated shell task is name *taskName*. The user data value is stored into the location pointed by *ppData*.

RETURNS **OK**, or **ERROR** if an error occured.

ERRNO **S_shellLib_NOT_SHELL_TASK**
taskName is not a shell task.

S_shellLib_NO_USER_DATA
Key *key* does not exist within the shell session context.

SEE ALSO **shellDataLib**, **shellDataGet( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataGet( )

NAME **shellDataGet( )** – get user data from a specified shell

SYNOPSIS

```
STATUS shellDataGet
    (
    SHELL_ID     shellId,  /* the shell identifier */
    const char * key,      /* the key for the value to get */
    void **      ppData    /* where to store the data value */
    )
```

DESCRIPTION This routine returns the data named *key* stored within the shell session *shellId*. The user data value is stored into the location pointed by *ppData*.

RETURNS **OK**, or **ERROR** if an error occured.

ERRNO **S_shellLib_NOT_SHELL_TASK**
taskName is not a shell task.

S_shellLib_NO_USER_DATA
the key name *key* does not exist within the shell session context *shellId*.

SEE ALSO **shellDataLib**, **shellFromNameDataGet( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataNext( )

NAME **shellDataNext( )** – get the next user data that matchs a key

SYNOPSIS

```
SHELL_ID shellDataNext
    (
    const char * key,      /* key data to search */
    SHELL_ID     shellId,  /* previous shell session id */
    void **      ppData    /* where to store data value */
    )
```

DESCRIPTION This routine returns the next user data named *key* in all shell session. *shellId* is the previous shell session checked. The user data value is stored into the location pointed by *ppData*.

RETURNS the shell session Id, or 0 if none shell session contains the key *key*.

ERRNO N/A

SEE ALSO **shellDataLib**, **shellDataFirst( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellDataRemove( )

NAME **shellDataRemove( )** – remove user data from a specified shell

SYNOPSIS

```
void shellDataRemove
    (
    SHELL_ID     shellId,  /* the shell identifier */
    const char * key,      /* the key for the value to remove */
    BOOL         finalize  /* TRUE to call the finalize routine */
    )
```

DESCRIPTION This routine removes the data defined by the key *key* from the shell session *shellId*.

If *finalize* is **TRUE**, the finalize routine associated with the data is called before the data is removed from the list.

RETURNS N/A

ERRNO N/A

SEE ALSO **shellDataLib**, **shellDataAdd( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellErrnoGet( )

NAME **shellErrnoGet( )** – get the shell session errno

SYNOPSIS

```
int shellErrnoGet
    (
    SHELL_ID shellId  /* shell session Id */
    )
```

DESCRIPTION This routine returns the errno value for the shell session *shellId*. The shell task errno is set to this value before calling a VxWorks function.

shellId can be equal to **CURRENT_SHELL_SESSION**.

RETURNS the errno value, or -1 if *shellId* is not a valid shell session

ERRNO N/A

SEE ALSO **shellLib**, **shellErrnoSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellErrnoSet( )

NAME **shellErrnoSet( )** – set the shell session errno

SYNOPSIS

```
void shellErrnoSet
    (
    SHELL_ID shellId,   /* shell session Id */
    int      errNo      /* errno number */
    )
```

DESCRIPTION This routine sets the errno value for the shell session *shellId* to *errNo*. The shell task errno is set to this value before calling a VxWorks function; when the VxWorks function returns, the current errno value of the shell task is save using **shellErrnoSet( )**.

shellId can be equal to **CURRENT_SHELL_SESSION**.

RETURNS N/A

ERRNO N/A

SEE ALSO **shellLib**, **shellErrnoGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellFirst( )

NAME **shellFirst( )** – get the first shell session

SYNOPSIS

```
SHELL_ID shellFirst (void)
```

DESCRIPTION This routine returns the Id of the first running shell session.

RETURNS the first shell Id, or 0 if no shell session is running.

ERRNO N/A

SEE ALSO **shellLib**, **shellNext( )**, **shellFromTaskGet( )**, **shellFromNameGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellFromNameGet( )

NAME **shellFromNameGet( )** – get a shell session Id from a task name

SYNOPSIS

```
SHELL_ID shellFromNameGet
    (
    const char * taskName  /* the shell task name */
    )
```

DESCRIPTION This routine returns the shell session Id of the shell task whose name is *taskName*.

RETURNS the shell session Id, or 0 if the task is not a shell task.

ERRNO N/A

SEE ALSO **shellLib**, **shellFirst( )**, **shellFromTaskGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellFromTaskGet( )

NAME **shellFromTaskGet( )** – get a shell session Id from its task Id

SYNOPSIS

```
SHELL_ID shellFromTaskGet
```

```
    (
    int taskId  /* the shell task ID or 0 */
    )
```

DESCRIPTION This routine returns the shell session identifier whose task identifier is *taskId*. If *taskId* is 0, the current task Id is used.

RETURNS the shell session identifier, or 0 if *taskId* is not a shell task.

ERRNO N/A

SEE ALSO **shellLib**, **shellFirst( )**, **shellFromNameGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellGenericInit( )

NAME **shellGenericInit( )** – start a shell session

SYNOPSIS

```
STATUS shellGenericInit
    (
    const char * config,       /* configuration string or NULL */
    int          stackSize,    /* shell stack (0 = default value) */
    const char * shellName,    /* shell task name or NULL for def. base name
*/
    char **      pShellName,   /* pointer on the shell task name or NULL */
    BOOL         interactive,  /* interactive mode if TRUE */
    BOOL         loginAccess,  /* login access */
    int          fdin,         /* input file descriptor */
    int          fdout,        /* output file descriptor */
    int          fderr         /* error file descriptor */
    )
```

DESCRIPTION This routine starts a shell session. This is a generic routine to start a shell session.

config is a string that holds the values of the configuration variables this new shell session. *stackSize* defines the size of the stack allocated for the shell task. A value of 0 is used to define the default value. *shellName* is a pointer on the desired shell task name. If this parameter is **NULL**, a generic name is used for each shell task. *pShellName*, if not **NULL**, will return a pointer on the shell task name. *interpName* is the desired interpreter name to use with this shell session. *interactive* is set to **TRUE** if this shell session is interactive; **FALSE** otherwise. *loginAccess* is set to **TRUE** if one wants the user to identify itself with a login and password before accessing the shell (this feature is only usefull if a login function has been previously installed). *fdin*, *fdout* and *fderr* are the file descriptors to use respectively for the standard input, output and error of the shell task.

If *loginAccess* is set to **TRUE**, and if the **INCLUDE_SECURITY** component is installed, the user will be asked for a login and a password before the shell start.

NOTE 1 If the shell is configured to use only one shell task for all the connections, (compatibility mode), this routine only changes the I/O of the only shell session, and restarts it. On termination of the shell task (see **shellTerminate( )**), the previous I/O of the shell task is restored, and the shell task is restarted.

NOTE 2 If the shell is configured to use only one shell task for all the connections, *pShellName* may return a task name different from the one specified by *shellName*.

RETURNS **OK**, or **ERROR** if the shell session cannot be created.

ERRNO

S_shellLib_NO_INTERP
The interpreter specified is not registered.

S_shellLib_SHELL_TASK_EXISTS
A shell session with the same name or the same standard input already exists.

S_shellLib_SHELL_TASK_MAX
The maximun number of shell session has been reached.

S_shellLib_INTERNAL_ERROR
An internal error prevents the shell session to be created.

taskSpawn( ) errnos, **malloc( )** and **memPartCreate( )** errno.

SEE ALSO **shellLib**, **shellLock( )**, **shellTerminate( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellHistory( )

NAME **shellHistory( )** – display or set the size of the shell history (vxWorks 5.5 compatibility)

SYNOPSIS

```
STATUS shellHistory
    (
    int size  /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This routine displays shell history, or resets the default number of commands displayed by shell history to *size*. By default, history size is 20 commands. Shell history is actually maintained by **ledLib**. If *size* is 0, the routine displays the line history.

IMPORTANT NOTE This routine is backward compatible with previous version of the kernel shell. It only changes history size of the current shell session. If the current task is not a shell task, the routine will use the shell task attached to the console (if one exists).

RETURNS **OK**, or **ERROR** if there is not a shell task attached to the console.

ERRNO N/A

SEE ALSO **shellLib**, **ledLib**, **ledControl( )**, **h( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**, *Wind River Workbench Command-Line User's Guide 2.2:* **Host Shell**

shellIdVerify( )

NAME **shellIdVerify( )** – verify the validity of a shell session Id

SYNOPSIS

```
STATUS shellIdVerify
    (
    SHELL_ID shellId  /* shell session Id to verify */
    )
```

DESCRIPTION This routine checks the validity of the shell session identifier *shellId*.

RETURNS **OK** if the shell identifier is valid, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **shellLib**, **shellFirst( )**, **shellFromNameGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpByNameFind( )

NAME **shellInterpByNameFind( )** – Find an interpreter based on its name

SYNOPSIS

```
SHELL_INTERP * shellInterpByNameFind
    (
    const char * interpName  /* shell interpreter name to find */
    )
```

DESCRIPTION This routine checks if an interpreter with name *interpName* does exist.

RETURNS a pointer on the interpreter structure if an interpreter is found, or **NULL** if the interpreter with specified name is not registered.

ERRNO N/A

SEE ALSO **shellInterpDefaultNameGet( )**, **shellInterpRegister( )**, **shellLib**, **shellInterpLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpCtxGet( )

NAME **shellInterpCtxGet( )** – get the interpreter context

SYNOPSIS

```
SHELL_INTERP_CTX * shellInterpCtxGet
    (
    SHELL_ID shellId  /* shell session identifier */
    )
```

DESCRIPTION This routine returns a pointer to the context of the current interpreter used by the shell session *shellId*

RETURNS a pointer to the interpreter context, or **NULL** if none interpreter is active.

ERRNO N/A

SEE ALSO **shellInterpLib**, **shellInterpNameGet( )**, **shellInterpRegister( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpDefaultNameGet( )

NAME **shellInterpDefaultNameGet( )** – get the name of the default interpreter

SYNOPSIS

```
const char * shellInterpDefaultNameGet (void)
```

DESCRIPTION This routine returns the name of the default interpreter.

RETURNS a pointer on the interpreter name, or **NULL** if none interpreter is registered.

ERRNO N/A

SEE ALSO **shellInterpLib**, **shellInterpNameGet( )**, **shellInterpRegister( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpEvaluate( )

NAME **shellInterpEvaluate( )** – interpret a string by an interpreter

SYNOPSIS

```
STATUS shellInterpEvaluate
    (
    char *             arg,              /* argument to evaluate */
    const char *       interpreterName,  /* or NULL for default */
    SHELL_EVAL_VALUE * pValue            /* interpreter return value */
    )
```

DESCRIPTION This routine interprets the string *arg* by the interpreter named *interpreterName*. The result value is returned in *pValue*. If *interpreterName* is **NULL**, the function uses the default interpreter, the one which was registered first. Note that the string *arg* may be modified by the interpreter, you need to save it before calling this routine if you want to use it later.

RETURNS **OK**, or **ERROR** if an error occured

ERRNO

S_shellLib_NO_INTERP
The interpreter specified is not registered or it does not have an evaluation routine.

S_shellLib_NOT_SHELL_TASK
The current task is not a shell task.

S_shellLib_INTERNAL_ERROR
An internal error occured and prevents the evaluation.

S_shellLib_INTERP_INIT_ERROR
The context of the interpreter cannot be initialized.

Interpreter's evaluation routine errnos.

SEE ALSO **shellInterpLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpNameGet( )

NAME **shellInterpNameGet( )** – get the name of the current interpreter

SYNOPSIS

```
const char * shellInterpNameGet
    (
    SHELL_ID shellId  /* shell session ID */
    )
```

DESCRIPTION This routine returns the name of the current interpreter of the shell session *shellId*.

RETURNS a pointer on the interpreter name, or **NULL** if none interpreter is defined

ERRNO N/A

SEE ALSO **shellInterpLib**, **shellInterpDefaultNameGet( )**, **shellInterpRegister( )**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellInterpRegister( )

NAME **shellInterpRegister( )** – register a new interpreter

SYNOPSIS

```
STATUS shellInterpRegister
    (
    FUNCPTR interpInitRtn  /* interpreter init routine */
    )
```

DESCRIPTION This routine is used to register a new interpreter against the kernel shell.

interpInitRtn is the initialization routine of the interpreter to register. Its definition is:

```
STATUS init
  (
  SHELL_INTERP *  pInterp        /* interpreter structure */
  )
```

with *pInterp* a pointer on an interpreter structure.

The *interpInitRtn* routine is called at registration time. This routine must complete the fields of the **SHELL_INTERP** structure with the addresses of the interpreter routines:

- the interpreter context initialization function,
- the parser function,
- the evaluation function,
- the completion function,
- the restart function (called whenever the shell is restarted, by CTRL-C key combination for example),
- the interpreter context finalizer function (to release any resources).

Any of these routine addresses (except the init function) can be **NULL**.

The *interpInitRtn* routine also has to set the interpreter name and default prompt. The interpreter name must be unique among the registered interpreters.

The definitions of the interpreter routines are:

```
STATUS ctxInit
```

```
    (
    SHELL_INTERP_CTX *  pInterpCtx    /* interpreter context */
    )
```

with *pInterpCtx* a pointer to the interpreter context, which is unique per shell session. This routine is called either when a new shell session is started, when a statement is evaluated or when the shell is switched to a new interpreter for which the interpreter context does not exist yet. This routine returns **ERROR** if an error occured, **OK** otherwise.

```
STATUS parser
    (
    SHELL_INTERP_CTX *  pInterpCtx,   /* interpreter context */
    const char *        inputLine,    /* input line to parse */
    BOOL                isInteractive /* TRUE for interactive session */
    )
```

with *pInterpCtx* a pointer to the interpreter context, *inputLine* a pointer to the input string to interpret. *isInteractive* is **TRUE** if the parsing is interactive. This routine returns **ERROR** if an error occured, **OK** otherwise.

```
STATUS evaluate
    (
    SHELL_INTERP_CTX *  pInterpCtx,   /* interpreter context */
    const char *        inputLine,    /* input line to parse */
    SHELL_EVAL_VALUE *  pValue        /* where to store return value */
    )
```

This routine is used to evaluate a string *inputLine* by the interpreter. The resulting value is stored into the shell value pointed by *pValue*. *pInterpCtx* is a pointer to the interpreter context. This routine returns **ERROR** if an error occured, **OK** otherwise.

```
STATUS completion
    (
    SHELL_INTERP_CTX *  pInterpCtx,   /* interpreter context */
    LED_ID              ledId,        /* LED identifier */
    char *              line,         /* line to complete */
    UINT                lineSize,     /* size of line buffer */
    UINT *              pCursorPos,   /* cursor position in the line */
    char                completionChar /* completion character */
    )
```

This routine is called when a completion character is typed on the input. The completion characters are defined by the Line EDiting mode used. *ledId* identifies the Line EDiting session. *line* is the input line currently printed. Its maximal size is *lineSize*. The terminal EOS is not counted with this size. *pCursorPos* is a pointer to the position of the cursor in *line*. *completionChar* is the character typed for completion. *pInterpCtx* is a pointer to the interpreter context. This routine returns **ERROR** if the completion does not succeed, **OK** otherwise.

```
void ctxRestart
  (
  SHELL_INTERP_CTX *  pInterpCtx      /* interpreter context */
  )
```

with *pInterpCtx* a pointer to the interpreter context. This routine is called whenever the shell session is restarted. It is up to the interpreter to release any allocated resources.

```
STATUS ctxFinalize
  (
  SHELL_INTERP_CTX *  pInterpCtx      /* interpreter context */
  )
```

with *pInterpCtx* a pointer to the interpreter context. This routine is used to free any internal resource used by the interpreter for a shell session. It is called when a shell session is terminated. This routine returns **ERROR** if an error occured, **OK** otherwise.

The *ctxInit* routine does not need to set up the **currentPrompt** field of the *pInterpCtx* structure, but only the **prompt** field with the prompt string of the interpreter.

The *pInterpParam* field of *pInterpCtx* can be used to store the address of an internal structure of the interpreter.

The first interpreter registered becomes the default interpreter. The default interpreter is used by a shell session if no name is defined for its initialisation.

RETURNS **OK**, or **ERROR** if an error occured.

ERRNO

S_shellLib_INTERP_EXISTS
An interpreter with the same name is already registered

S_shellLib_INTERNAL_ERROR
An internal error occurs. The interpreter is not registered.

memPartAlloc( ), **malloc( )** errnos

SEE ALSO **shellInterpLib**, **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellLock( )

NAME **shellLock( )** – lock access to the shell (vxWorks 5.5 compatibility)

SYNOPSIS

```
BOOL shellLock
    (
    BOOL request  /* TRUE = lock, FALSE = unlock */
    )
```

VxWorks 5.5 behavior

This routine locks or unlocks access to the shell. When locked, cooperating tasks, such as **telnetdTask( )** and **rlogindTask( )**, will not be able to control the shell.

If the shell is configured to use a unique task (compatibility mode), this routine reacts as the VxWorks 5.5 version. But for normal mode of the shell (multiple shell sessions), the routine always returns **TRUE**.

RETURNS If the shell session is unique, **TRUE** if *request* is "lock" and the routine successfully locks the shell, otherwise **FALSE**; **TRUE** if request is "unlock" and the routine successfully unlocks the shell, otherwise **FALSE**. In multiple shell session mode, **TRUE** is always returned.

ERRNO N/A

SEE ALSO **shellLib**, **shellCompatibleSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellNext( )

NAME **shellNext( )** – get the next shell session

SYNOPSIS

```
SHELL_ID shellNext
    (
    SHELL_ID shellId  /* shell ID whose successor is to be found */
    )
```

DESCRIPTION This routine returns the Id of the next shell session, compared to *shellId*.

RETURNS the next shell Id or 0 if there is no more running shell.

ERRNO N/A

SEE ALSO **shellLib**, **shellFirst( )**, **shellFromTaskGet( )**, **shellFromNameGet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellPromptFmtDftSet( )

NAME **shellPromptFmtDftSet( )** – set the default prompt format string

SYNOPSIS

```
STATUS shellPromptFmtDftSet
    (
    const char * interp,    /* interpreter name or NULL for default */
    const char * promptFmt  /* prompt format string or NULL for initial */
    )
```

DESCRIPTION This routine sets the format of the default prompt of the interpreter named *interp* to *promptFmt*. If *interp* is **NULL**, the default interpreter is used (the first registered). If *promptFmt* is **NULL**, the initial interpreter prompt, the one defined by the interpreter init function, is restored.

NOTE Setting the default prompt of an interpreter after its first use will not modify the current prompt for a shell session. The change will be visible for new shell sessions or if it is done prior the first use of the interpreter. In order to change the current prompt, the routine **shellPromptFmtSet( )** should be used.

RETURNS **OK**, or **ERROR** if the memory for the prompt cannot be allocated or if the interpreter is not registered.

ERRNO **S_shellLib_NO_INTERP**
none interpreter named *interp* exists.

malloc( ) errnos, **memPartCreate( )** errnos.

SEE ALSO **shellPromptLib**, **shellPromptFmtSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellPromptFmtSet( )

NAME **shellPromptFmtSet( )** – set the current prompt format string

SYNOPSIS

```
STATUS shellPromptFmtSet
    (
    SHELL_ID     shellId,   /* CURRENT_SHELL_SESSION for current shell */
    const char * interp,    /* interpreter name or NULL for current */
    const char * promptFmt  /* prompt format string or NULL for default */
    )
```

DESCRIPTION This routine sets the format of the current prompt of the interpreter named *interp*, associated with the shell session *shellId*, to *promptFmt*. If *shellId* is **CURRENT_SHELL_SESSION**, the current shell session is used. If *interp* is **NULL**, the current interpreter of *shellId* is used. If *promptFmt* is **NULL**, the default interpreter prompt is restored.

NOTE Setting the current prompt for a shell session does not affect the other running sessions or new shell sessions. To change the default prompt of an interpreter, the routine **shellPromptFmtDftSet( )** should be used instead.

RETURNS **OK**, or **ERROR** if the interpreter does not exist or a memory error occured.

ERRNO **S_shellLib_NOT_SHELL_TASK**
shellId is an invalid shell session.

S_shellLib_NO_INTERP
none interpreter named *interp* exists.

malloc( ) errnos, **memPartCreate( )** errnos.

SEE ALSO **shellPromptLib**, **shellPromptFmtDftSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellPromptFmtStrAdd( )

NAME **shellPromptFmtStrAdd( )** – add a new prompt format string

SYNOPSIS

```
STATUS shellPromptFmtStrAdd
    (
    char        fmt,      /* format character */
    VOIDFUNCPTR fmtRtn,   /* format routine */
    BOOL        force     /* TRUE to superseed a previous definition */
    )
```

DESCRIPTION This routine adds a new format string which manage the format character *fmt*. The display function associated with the format is *fmtRtn*. If *force* is **TRUE**, the format routine *fmtRtn* will superseed a previous definition of the format routine.

When the shell prompt is printed, each format routine associated with the format strings used in the prompt string are called with the prototype:

```
void fmtRtn
    (
    SHELL_ID shellId     /* shell session Id */
    )
```

with *shellId* the identifier of the shell session managed. The format routine can use regular output functions (**printf( )**, **printErr( )** ...) as the IO of the current task is correctly redirected to the dedicated shell terminal.

RETURNS **OK**, or **ERROR** if the format cannot be added.

ERRNO

SEE ALSO **shellPromptLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellPromptSet( )

NAME **shellPromptSet( )** – change the shell prompt (vxWorks 5.5 compatibility)

SYNOPSIS

```
STATUS shellPromptSet
```

```
    (
    const char * newPrompt  /* string to become new shell prompt */
    )
```

DESCRIPTION This routine changes the format of the current shell prompt of the C interpreter to *newPrompt*. If *newPrompt* is **NULL**, the default C interpreter prompt is restored instead.

IMPORTANT NOTE This routine is backward compatible with previous version of the kernel shell. This routine changes the prompt of the C interpreter only. If the shell is configured as compatible with VxWorks 5.5 (**SHELL_COMPATIBLE** parameter is **TRUE**), the default C interpreter prompt is also modified, not only the current prompt. If the current task is not a shell task, the routine will use the shell task attached to the console (if one exists).

RETURNS **OK**, or **ERROR** if there is not a shell task attached to the console.

ERRNO N/A

SEE ALSO **shellLib**, **shellPromptFmtSet( )**, **shellPromptFmtDftSet( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**, *Wind River Workbench Command-Line User's Guide 2.2:* **Host Shell**

shellResourceReleaseHookAdd( )

NAME **shellResourceReleaseHookAdd( )** – add a resource-releasing hook to the shell

SYNOPSIS

```
STATUS shellResourceReleaseHookAdd
    (
    SHELL_RES_RELEASE_HOOK hook  /* hook routine to add */
    )
```

DESCRIPTION The function registers a routine which will be run when the shell is terminated or aborted. The routine is meant to release mutexes, semaphores or related data structures used within the shell components. When an abnormal shell exit occurs, this is used for deadlock prevention.

Each hook routine must be of the following form :

```
void releaseHook
    (
    SHELL_ID  shellId,         /* shell session Id */
    BOOL      force            /* whether to force-release */
    )
```

The hooks are meant to release global resources, otherwise the release would be implemented in context-related termination routines. However, it is possible that session-related operations be needed. For that purpose, the hook takes a session ID as a parameter.

The *force* parameter is used to indicate that force-relinquishing is needed. This is because the context in which the hook executes might not be the one of the resource owner. For instance, **semMForceGive( )** is be used instead of **semGive( )** in that case.

RETURNS **OK**, or **ERROR** if the hook table is full or an error occurred.

ERRNO

SEE ALSO **shellLib**

shellRestart( )

NAME **shellRestart( )** – restart a shell session

SYNOPSIS

```
STATUS shellRestart
    (
    SHELL_ID shellId  /* the ID of the shell session to restart */
    )
```

DESCRIPTION This routine restarts the shell session *shellId*.

shellId can be a valid session identifier or **CURRENT_SHELL_SESSION**.

RETURNS **OK**, or **ERROR** it not possible to restart the shell session.

ERRNO N/A

SEE ALSO **shellLib**, **shellTaskGet( )**, **taskRestart( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellScriptAbort( )

NAME **shellScriptAbort( )** – signal the shell to stop processing a script (vxWorks 5.5 compatibility)

SYNOPSIS

```
STATUS shellScriptAbort (void)
```

DESCRIPTION This routine signals the current shell session to abort processing a script file. It can be called from within a script if an error is detected.

IMPORTANT NOTE This routine is kept for backward compatibility reason with previous version of the kernel shell. This routine aborts scripting of the current shell session. If the current task is not a shell task, the function will use the shell task attached to the console (if one exists).

RETURNS **OK**, or **ERROR** if there is not a shell task attached to the console.

ERRNO N/A

SEE ALSO **shellLib**, **shellScriptNoAbort( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellTaskGet( )

NAME **shellTaskGet( )** – get the task Id of a shell session

SYNOPSIS

```
int shellTaskGet
    (
    SHELL_ID shellId  /* shell session Id */
    )
```

DESCRIPTION This routine returns the task Id for the shell session *shellId*.

RETURNS the shell task Id, or **ERROR** if the shell session is not valid.

ERRNO N/A

SEE ALSO **shellLib**, **shellIdVerify( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellTaskIdDefault( )

NAME **shellTaskIdDefault( )** – set the default task for a given shell session

SYNOPSIS

```
int shellTaskIdDefault
    (
    SHELL_ID shellId,  /* shell session Id */
    int      taskId    /* default task Id or 0 */
    )
```

DESCRIPTION This routine set the default task associated with the shell session *shellId* to *taskId*.

If *shellId* is equal to **CURRENT_SHELL_SESSION**, the current shell session is used. If this function is called outside of a shell session, and *shellId* is **CURRENT_SHELL_SESSION**, the shell session attached to the console is used (if one exists).

If *shellId* is equal to **ALL_SHELL_SESSIONS**, the default task *taskId* is set for all shell sessions.

If *taskId* is 0, the default task of the shell session *shellId* is returned.

RETURNS the last default task Id (may be 0 if not previously set)

ERRNO N/A

SEE ALSO **shellLib**, **taskIdDefault( )**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shellTerminate( )

NAME **shellTerminate( )** – terminate a shell task

SYNOPSIS

```
void shellTerminate
    (
    SHELL_ID shellId  /* the shell ID to terminate */
    )
```

DESCRIPTION This routine kills a shell session, based on its Id *shellId*. Suicide is prohibited (a shell session cannot kill itself by this call; use **exit( )** instead).

NOTE If the shell is configured to use only one shell session for all the connections, (compatible mode), this routine only restores the I/O of the shell task (see **shellGenericInit( )**), and then restarts it.

RETURNS N/A

ERRNO N/A

SEE ALSO **shellLib**, *VxWorks Kernel Programmer's Guide:* **Kernel Shell**

shlShow( )

NAME

shlShow( ) – display information for shared libraries

SYNOPSIS

```
BOOL shlShow
    (
    SHL_ID shlId,  /* Shared Lib ID */
    int    level   /* 0 = summary, 1 = detailed, 2 = all */
    )
```

DESCRIPTION

This routine displays information for shared libraries (SHL). This routine takes two parameters, *shlId* and *level*. The first parameter is the shared library ID, which may be obtained by using **rtpShlShow( )**. The second parameter is the level of detail to display the SHL information.

Depending on the level and the SHL ID specified, the information displayed differs. If the *level* is 0, then it displays the summary information for either the specified SHL or all SHLs in the system.   If the *level* is 1, then **shlShow( )** displays the detailed information for the specified SHL. If *level* is 2, **shlShow( )** displays the detailed information for all SHLs in the system, regardless of the SHL ID you specify. Refer to the table for more information.

Level	SHL ID	Meaning
0	0	Display summary information for all SHLs.
0	SHL	Display summary information for specified SHL.
1	0	Invalid Display of the SHL, must specify SHL.
1	SHL	Display detailed information for specified SHL.
2	ANY	Display detailed information for all SHLs.

shlShow( ) only displays the SHL name up to a maximum of 20 characters long. If the name is more than 20 characters, the name will be truncated to 18 characters for displaying purposes. Prepended to the truncated name, a "<" will be display to indicate that the name is more than 20 characters long. To get a display of the full SHL name, display the SHL with the *level* set to 1.

For the command interpreter shell, use the **shl** and the **shl info** commands to display SHL information.

Use this routine to display information on SHLs in the system. For information on SHLs associated with an RTP, use the **rtpShlShow( )** routine to display this information.

EXAMPLE

Below is an example display of a shared library.

```
-> shlShow 0x11c0724

      SHL NAME           ID      TEXT ADDR  TEXT SIZE  DATA SIZE  REF CNT
-------------------- ---------- ---------- ---------- ---------- -------
< tty/slDfw/libSo.so          1 0xff435000      0x574      0x628    2
```

RETURNS

TRUE, or **FALSE** if the SL is invalid.

ERRNO

Possible errnos generated by this function include:

S_objLib_OBJ_ID_ERROR
An incorrect SHL ID was provided.

S_objLib_ACCESS_DENIED
Unable to get exclusive access to the SHL list.

SEE ALSO

shlShow, **rtpLib**, **rtpShow**, **shlLib**, the VxWorks programmer guides.

shlSymsAdd( )

NAME

shlSymsAdd( ) – add symbols from a shared object file to a RTP symbol table

SYNOPSIS

```
STATUS shlSymsAdd
    (
    void * shlId,       /* ID of the shared library */
    RTP_ID rtpId,       /* RTP the symbols should be added to */
    UINT32 regPolicy,   /* symbol registration policy */
    char * filePath     /* path and name of the shared object file */
    )
```

DESCRIPTION

This command is provided as a help in case a RTP needs to be debugged but has been launched with an empty symbol table. It forces the registration of the symbols from a shared object file into a RTP symbol table.

Note that this command does not verify whether the symbols are already in the symbol table and does not prevent the creation of multiple occurences of these symbols.

It is important to understand that symbols are added to the symbol table in the order of their registration and that the most recent entry will hide symbols of same name already registered. The **rtpLkup( )** command will show all occurences of the symbols of a given name so it is possible to use their addresses instead of their names if there is a risk of confusion.

The only required information are the shared library ID (*shlId* parameter) and the RTP ID (*rtpId* parameter).

The *regPolicy* parameter sets the symbol registration policy. The policy can be one of the following:

0x01 (**RTP_GLOBAL_SYMBOLS**)
Add only global symbols to the symbol table. This is the default when the parameter is left null.

0x02 (**RTP_LOCAL_SYMBOLS**)
Add only local symbols to the symbol table.

0x03 (**RTP_ALL_SYMBOLS**)
Add both local and global symbols to the symbol table.

The *filePath* parameter overrides the path recorded for the shared library. It may be left null if the symbols should be read from the same file as the one used to create the shared library with. This parameter must be used when the symbols should be read from a file stored in a different location than what was recorded when the shared library has been created.

RETURNS **OK** if the symbols could be read and recorded, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **usrRtpLib**, **shlSymsRemove( )**, **rtpSymsAdd( )**, **rtpSymsRemove( )**

shlSymsRemove( )

NAME **shlSymsRemove( )** – remove shared library symbols from a RTP symbol table

SYNOPSIS

```
STATUS shlSymsRemove
    (
    void * shlId,     /* ID of the shared library */
    RTP_ID rtpId,     /* RTP the symbols should be removed from */
    UINT32 remPolicy  /* symbol removal policy */
    )
```

DESCRIPTION This command forces the removal of symbols related to a shared library from a RTP symbol table.

The *remPolicy* parameter sets the symbol removal policy. The policy can be one of the following:

0x02 (**RTP_LOCAL_SYMBOLS**)
Remove only the shared library's local symbols from the symbol table.

0x03 (**RTP_ALL_SYMBOLS**)
Removes both the shared library's local and global symbols from the symbol table.

RETURNS **OK** if the symbols could be removed, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **usrRtpLib**, **shlSymsAdd( )**, **rtpSymsAdd( )**, **rtpSymsRemove( )**

show( )

NAME **show( )** – print information on a specified object

SYNOPSIS

```
void show
    (
    int objId,  /* object ID */
    int level   /* information level */
    )
```

DESCRIPTION This command prints information on the specified object. System objects include tasks, local and shared semaphores, local and shared message queues, local and shared memory partitions, watchdogs, and symbol tables. An information level is interpreted by the objects show routine on a class by class basis. Refer to the object's library manual page for more information.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **i( )**, **ti( )**, **lkup( )**, the VxWorks programmer guides.

sigInit( )

NAME **sigInit( )** – initialize the signal facilities

SYNOPSIS

```
int sigInit
    (
    BOOL posixMode
    )
```

DESCRIPTION This routine initializes the signal facilities. It is usually called from the system start-up routine **usrInit( )** in usrConfig, before interrupts are enabled.

If the boolean parameter *posixMode* is **TRUE** then the signals sent to a faulting task will be POSIX conformant, if it is **FALSE** the signals will be backwards compatible with previous versions of VxWorks.

RETURNS **OK**, or **ERROR** if the delete hooks cannot be installed.

ERRNO **S_taskLib_TASK_HOOK_TABLE_FULL**
Task hook table is full and signal delete hook can not be added.

SEE ALSO **sigLib**

sigaction( )

NAME **sigaction( )** – examine and/or specify the action associated with a signal (POSIX)

SYNOPSIS

```
int sigaction
    (
    int                     signo,  /* signal of handler of interest */
    const struct sigaction *pAct,   /* location of new handler */
    struct sigaction       *pOact   /* location to store old handler */
    )
```

DESCRIPTION This routine allows the calling process to examine and/or specify the action to be associated with a specific signal.

RETURNS **OK** (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**

sigaddset( )

NAME **sigaddset( )** – add a signal to a signal set (POSIX)

SYNOPSIS

```
int sigaddset
    (
    sigset_t *pSet,  /* signal set to add signal to */
    int      signo   /* signal to add */
    )
```

DESCRIPTION This routine adds the signal specified by *signo* to the signal set specified by *pSet*.

RETURNS **OK** (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**

sigblock( )

NAME

sigblock( ) – add to a set of blocked signals

SYNOPSIS

```
int sigblock
    (
    int mask  /* mask of additional signals to be blocked */
    )
```

DESCRIPTION

This routine adds the signals in *mask* to the task's set of blocked signals. A one (1) in the bit mask indicates that the specified signal is blocked from delivery. Use the macro SIGMASK to construct the mask for a specified signal number.

This routine has been deprecated, instead use **sigprocmask( )**.

RETURNS

The previous value of the signal mask.

ERRNO

N/A

SEE ALSO

sigLib, **sigprocmask( )**

sigdelset( )

NAME

sigdelset( ) – delete a signal from a signal set (POSIX)

SYNOPSIS

```
int sigdelset
    (
    sigset_t *pSet,  /* signal set to delete signal from */
    int      signo   /* signal to delete */
    )
```

DESCRIPTION

This routine deletes the signal specified by *signo* from the signal set specified by *pSet*.

RETURNS

OK (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib

sigemptyset( )

NAME **sigemptyset( )** – initialize a signal set with no signals included (POSIX)

SYNOPSIS

```
int sigemptyset
    (
    sigset_t *pSet  /* signal set to initialize */
    )
```

DESCRIPTION This routine initializes the signal set specified by *pSet*, such that all signals are excluded.

RETURNS **OK** (0), or **ERROR** (-1) if the signal set cannot be initialized.

ERRNO N/A

SEE ALSO **sigLib**

sigfillset( )

NAME **sigfillset( )** – initialize a signal set with all signals included (POSIX)

SYNOPSIS

```
int sigfillset
    (
    sigset_t *pSet  /* signal set to initialize */
    )
```

DESCRIPTION This routine initializes the signal set specified by *pSet*, such that all signals are included.

RETURNS **OK** (0), or **ERROR** (-1) if the signal set cannot be initialized.

ERRNO N/A

SEE ALSO **sigLib**

sigismember( )

NAME **sigismember( )** – test to see if a signal is in a signal set (POSIX)

SYNOPSIS

```
int sigismember
```

```
    (
    const sigset_t *pSet,  /* signal set to test */
    int            signo   /* signal to test for */
    )
```

DESCRIPTION This routine tests whether the signal specified by *signo* is a member of the set specified by *pSet*.

RETURNS 1 if the specified signal is a member of the specified set, **OK** (0) if it is not, or **ERROR** (-1) if the test fails.

ERRNO **EINVAL**

SEE ALSO **sigLib**

signal( )

NAME **signal( )** – specify the handler associated with a signal

SYNOPSIS

```
void (*signal
    (
    int                 signo,
    void  (*pHandler) ()
    )) ()
```

DESCRIPTION This routine chooses one of three ways in which receipt of the signal number *signo* is to be subsequently handled. If the value of *pHandler* is **SIG_DFL**, default handling for that signal will occur. If the value of *pHandler* is **SIG_IGN**, the signal will be ignored. Otherwise, *pHandler* must point to a function to be called when that signal occurs.

A signal handler associated with *signo* as a result of a call to this routine will be reset to **SIG_DFL** upon entry into the signal handler. Subsequent instances of *signo* will thus be handled with the default action. The **sigaction( )** routine must be used if this behavior is not desired.

RETURNS The value of the previous signal handler, or **SIG_ERR**.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **sigaction( )**

sigpending( )

NAME

sigpending( ) – retrieve the set of pending signals blocked from delivery (POSIX)

SYNOPSIS

```
int sigpending
    (
    sigset_t *pSet  /* location to store pending signal set */
    )
```

DESCRIPTION

This routine stores the set of signals that are blocked from delivery and that are pending for the calling process in the space pointed to by *pSet*.

RETURNS

OK (0), or **ERROR** (-1) if the signal TCB cannot be allocated.

ERRNO

ENOMEM

SEE ALSO

sigLib

sigprocmask( )

NAME

sigprocmask( ) – examine and/or change the signal mask (POSIX)

SYNOPSIS

```
int sigprocmask
    (
    int             how,    /* how signal mask will be changed */
    const sigset_t *pSet,   /* location of new signal mask */
    sigset_t       *pOset   /* location to store old signal mask */
    )
```

DESCRIPTION

This routine allows the calling process to examine and/or change its signal mask. If the value of *pSet* is not **NULL**, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicates the manner in which the set is changed and consists of one of the following, defined in **signal.h**:

SIG_BLOCK
the resulting set is the union of the current set and the signal set pointed to by *pSet*.

SIG_UNBLOCK
the resulting set is the intersection of the current set and the complement of the signal set pointed to by *pSet*.

SIG_SETMASK
the resulting set is the signal set pointed to by *pSet*.

RETURNS **OK** (0), or **ERROR** (-1) if *how* is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **sigsetmask( )**, **sigblock( )**

sigqueue( )

NAME **sigqueue( )** – send a queued signal to a task

SYNOPSIS

```
int sigqueue
    (
    int                 tid,
    int                 signo,
    const union sigval  value
    )
```

DESCRIPTION The function **sigqueue( )** sends the signal specified by *signo* with the signal-parameter value specified by *value* to the process specified by *tid*.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**

EAGAIN

SEE ALSO **sigLib**, **taskSigqueue( )**

sigqueueInit( )

NAME **sigqueueInit( )** – initialize the queued signal facilities

SYNOPSIS

```
int sigqueueInit
    (
    int nQueues
    )
```

DESCRIPTION This routine initializes the queued signal facilities. It must be called before any call to **sigqueue( )**. It is usually called from the system start-up routine **usrInit( )** in usrConfig, after **sysInit( )** is called.

It allocates *nQueues* buffers to be used by **sigqueue( )**. A buffer is used by each call to **sigqueue( )** and freed when the signal is delivered (thus if a signal is block, the buffer is unavailable until the signal is unblocked.)

RETURNS **OK**, or **ERROR** if memory could not be allocated.

ERRNO N/A

SEE ALSO **sigLib**

sigsetmask( )

NAME **sigsetmask( )** – set the signal mask

SYNOPSIS

```
int sigsetmask
    (
    int mask  /* new signal mask */
    )
```

DESCRIPTION This routine sets the calling task's signal mask to a specified value. A one (1) in the bit mask indicates that the specified signal is blocked from delivery. Use the macro SIGMASK to construct the mask for a specified signal number.

This routine has been deprecated, instead use **sigprocmask( )**.

RETURNS The previous value of the signal mask.

ERRNO N/A

SEE ALSO **sigLib**, **sigprocmask( )**

sigsuspend( )

NAME **sigsuspend( )** – suspend the task until delivery of a signal (POSIX)

SYNOPSIS

```
int sigsuspend
    (
    const sigset_t *pSet  /* signal mask while suspended */
    )
```

DESCRIPTION
This routine suspends the task until delivery of a signal. While suspended, *pSet* is used as the set of masked signals.

NOTE
Since the **sigsuspend( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS
-1, always.

ERRNO
EINTR

SEE ALSO
sigLib

sigtimedwait( )

NAME
sigtimedwait( ) – wait for a signal

SYNOPSIS
```
int sigtimedwait
    (
    const sigset_t        *pSet,     /* the signal mask while suspended */
    siginfo_t             *pInfo,    /* return value */
    const struct timespec *pTimeout
    )
```

DESCRIPTION
The function **sigtimedwait( )** selects the pending signal from the set specified by *pSet*. If multiple signals in *pSet* are pending, it will remove and return the lowest numbered one. If no signal in *pSet* is pending at the time of the call, the task will be suspend until one of the signals in *pSet* become pending, it is interrupted by an unblocked caught signal, or until the time interval specified by *pTimeout* has expired. If *pTimeout* is **NULL**, then the timeout interval is forever.

If the *pInfo* argument is non-**NULL**, the selected signal number is stored in the **si_signo** member, and the cause of the signal is stored in the **si_code** member. If the signal is a queued signal, the value is stored in the **si_value** member of *pInfo*; otherwise the content of **si_value** is undefined.

The following values are defined in **signal.h** for **si_code**:

SI_USER
the signal was sent by the **kill( )** function.

SI_QUEUE
the signal was sent by the **sigqueue( )** function.

SI_TIMER
the signal was generated by the expiration of a timer set by **timer_settime( )**.

SI_ASYNCIO
the signal was generated by the completion of an asynchronous I/O request.

SI_MESGQ
the signal was generated by the arrival of a message on an empty message queue.

The function **sigtimedwait( )** provides a synchronous mechanism for tasks to wait for asynchromously generated signals. A task should use **sigprocmask( )** to block any signals it wants to handle synchronously and leave their signal handlers in the default state. The task can then make repeated calls to **sigtimedwait( )** to remove any signals that are sent to it.

RETURNS
Upon successful completion (that is, one of the signals specified by *pSet* is pending or is generated) **sigtimedwait( )** will return the selected signal number. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRNO
EINTR
The wait was interrupted by an unblocked, caught signal.

EAGAIN
No signal specified by *pSet* was delivered within the specified timeout period.

EINVAL
The *pTimeout* argument specified a **tv_nsec** value less than zero or greater than or equal to 1000 million.

SEE ALSO
sigLib, **sigwait( )**

sigvec( )

NAME
sigvec( ) – install a signal handler

SYNOPSIS
```
int sigvec
    (
    int                  sig,    /* signal to attach handler to */
    const struct sigvec *pVec,   /* new handler information */
    struct sigvec       *pOvec   /* previous handler information */
    )
```

DESCRIPTION
This routine binds a signal handler routine referenced by *pVec* to a specified signal *sig*. It can also be used to determine which handler, if any, has been bound to a particular signal: **sigvec( )** copies current signal handler information for *sig* to *pOvec* and does not install a signal handler if *pVec* is set to **NULL** (0).

Both *pVec* and *pOvec* are pointers to a structure of type **struct sigvec**. The information passed includes not only the signal handler routine, but also the signal mask and additional option bits. The structure **sigvec** and the available options are defined in **signal.h**.

RETURNS **OK** (0), or **ERROR** (-1) if the signal number is invalid or the signal TCB cannot be allocated.

ERRNO **EINVAL**

ENOMEM

SEE ALSO **sigLib**

sigwait( )

NAME **sigwait( )** – wait for a signal to be delivered (POSIX)

SYNOPSIS

```
int sigwait
    (
    const sigset_t *pSet,
    int            *pSig
    )
```

DESCRIPTION This routine waits until one of the signals specified in *pSet* is delivered to the calling thread. It then stores the number of the signal received in the the location pointed to by *pSig*.

The signals in *pSet* must not be ignored on entrance to **sigwait( )**. If the delivered signal has a signal handler function attached, that function is not called.

RETURNS **OK**, or **ERROR** on failure.

ERRNO N/A

SEE ALSO **sigLib**, **sigtimedwait( )**

sigwaitinfo( )

NAME **sigwaitinfo( )** – wait for real-time signals

SYNOPSIS

```
int sigwaitinfo
    (
    const sigset_t *pSet,  /* the signal mask while suspended */
    siginfo_t      *pInfo  /* return value */
    )
```

DESCRIPTION The function **sigwaitinfo( )** is equivalent to calling **sigtimedwait( )** with *pTimeout* equal to **NULL**. See that reference entry for more information.

RETURNS

Upon successful completion (that is, one of the signals specified by *pSet* is pending or is generated) **sigwaitinfo( )** returns the selected signal number. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRNO

EINTR
The wait was interrupted by an unblocked, caught signal.

SEE ALSO

sigLib

sil31xxBIST( )

NAME

sil31xxBIST( ) – Controller Built-In Self Test...

SYNOPSIS

```
STATUS sil31xxBIST
    (
    int ctrlNum
    )
```

DESCRIPTION

/NOMANUAL

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

vxbSI31xxStorage

sil31xxBISTShow( )

NAME

sil31xxBISTShow( ) – Show the results of the power-on BIST

SYNOPSIS

```
VOID sil31xxBISTShow
    (
    )
```

DESCRIPTION

none

RETURNS

Nothing

ERRNO

Not Available

SEE ALSO **vxbSI31xxStorage**

sil31xxDiskPresent( )

NAME **sil31xxDiskPresent( )** – Return **OK** if disk exists.

SYNOPSIS
```
STATUS sil31xxDiskPresent
    (
    int ctrlNum,
    int devNum
    )
```

DESCRIPTION none

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **vxbSI31xxStorage**

sil31xxDrvVxbInit( )

NAME **sil31xxDrvVxbInit( )** – Initialize the driver.

SYNOPSIS
```
void sil31xxDrvVxbInit
    (
    BUS_DEVICE_ID pDev,     /* vxbus DeviceID */
    int           ctrlNum  /* assigned instance (controller number) */
    )
```

DESCRIPTION Initialize the driver structure for a single instance of the controller. This routine would get called once for each 31xx device.

RETURNS N/A

ERRNO

SEE ALSO **vxbSI31xxStorage**

sil31xxIsr( )

NAME **sil31xxIsr( )** – Interrupt service routine.

SYNOPSIS

```
VOID sil31xxIsr
    (
    int arg
    )
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbSI31xxStorage**

sil31xxRegisterPortCallback( )

NAME **sil31xxRegisterPortCallback( )** – register the port call back for a PHYRdyChg

SYNOPSIS

```
STATUS sil31xxRegisterPortCallback
    (
    int         ctrlNum,
    int         portNum,
    VOIDFUNCPTR myCallbackPtr,
    VOID        *myParam
    )
```

DESCRIPTION event

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **vxbSI31xxStorage**

sil31xxSectorRW( )

NAME

sil31xxSectorRW( ) – read a single sector

SYNOPSIS

```
STATUS sil31xxSectorRW
    (
    int       ctrl,
    int       port,
    sector_t sector,
    uint32_t numSecs,
    char      *data,
    BOOL      isRead
    )
```

DESCRIPTION

This routine is called to read a single sector and dump the output on the console.

ctrl controller number *port* port number *sector* starting sector for I/O operation *numSecs* number of sectors to read *data* pointer to data buffer

RETURNS

OK, or **ERROR** if the parameters are not valid.

ERRNO

Not Available

SEE ALSO

vxbSI31xxStorage

sil31xxXbdCreate( )

NAME

sil31xxXbdCreate( ) – Create an XBD for the specified port.

SYNOPSIS

```
device_t sil31xxXbdCreate
    (
    int  ctrlNum,
    int  devNum,
    char *devName  /* NULL for default, override with value */
    )
```

DESCRIPTION

none

RETURNS

OK or **ERROR**

ERRNO

Not Available

SEE ALSO

vxbSI31xxStorage

sil31xxXbdDelete( )

NAME **sil31xxXbdDelete( )** – Delete an XBD for a specified port

SYNOPSIS

```
STATUS sil31xxXbdDelete
    (
    int ctrlNum,
    int devNum
    )
```

DESCRIPTION none

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **vxbSI31xxStorage**

sincos( )

NAME **sincos( )** – compute both a sine and cosine

SYNOPSIS

```
void sincos
    (
    double x,           /* angle in radians */
    double *sinResult,  /* sine result buffer */
    double *cosResult   /* cosine result buffer */
    )
```

DESCRIPTION This routine computes both the sine and cosine of *x* in double precision. The sine is copied to *sinResult* and the cosine is copied to *cosResult*.

RETURNS N/A

ERRNO Not Available

SEE ALSO **mathALib**

sincosf( )

NAME
sincosf( ) – compute both a sine and cosine

SYNOPSIS

```
void sincosf
    (
    float x,            /* angle in radians */
    float *sinResult,   /* sine result buffer */
    float *cosResult    /* cosine result buffer */
    )
```

DESCRIPTION
This routine computes both the sine and cosine of *x* in single precision. The sine is copied to *sinResult* and the cosine is copied to *cosResult*. The angle *x* is expressed in radians.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
mathALib

sinf( )

NAME
sinf( ) – compute a sine (ANSI)

SYNOPSIS

```
float sinf
    (
    float x  /* angle in radians */
    )
```

DESCRIPTION
This routine returns the sine of *x* in single precision. The angle *x* is expressed in radians.

RETURNS
The single-precision sine of *x*.

ERRNO
Not Available

SEE ALSO
mathALib

sinhf( )

NAME
sinhf( ) – compute a hyperbolic sine (ANSI)

SYNOPSIS
```
float sinhf
    (
    float x  /* number whose hyperbolic sine is required */
    )
```

DESCRIPTION
This routine returns the hyperbolic sine of *x* in single precision.

RETURNS
The single-precision hyperbolic sine of *x*.

ERRNO
Not Available

SEE ALSO
mathALib

sleep( )

NAME
sleep( ) – delay for a specified amount of time

SYNOPSIS
```
unsigned int sleep
    (
    unsigned int secs
    )
```

DESCRIPTION
This routine causes the calling task to be blocked for *secs* seconds.

The time the task is blocked for may be longer than requested due to the rounding up of the request to the timer's resolution or to other scheduling activities (e.g., a higher priority task intervenes).

RETURNS
Zero if the requested time has elapsed, or the number of seconds remaining if it was interrupted.

ERRNO
EINVAL

EINTR

SEE ALSO
timerLib, **nanosleep( )**, **taskDelay( )**

smMemAddToPool( )

NAME

smMemAddToPool( ) – add memory to shared memory system partition (VxMP Option)

SYNOPSIS

```
STATUS smMemAddToPool
    (
    char *   pPool,     /* pointer to memory pool */
    unsigned poolSize  /* block size in bytes */
    )
```

DESCRIPTION

This routine adds memory to the shared memory system partition after the initial allocation of memory. The memory added need not be contiguous with memory previously assigned, but it must be in the same address space.

pPool is the global address of shared memory added to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory added to the partition.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if access to the shared memory system partition fails.

ERRNO

S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib

smMemCalloc( )

NAME

smMemCalloc( ) – allocate memory for array from shared memory system partition (VxMP Option)

SYNOPSIS

```
void * smMemCalloc
    (
    int elemNum,  /* number of elements */
    int elemSize  /* size of elements */
    )
```

DESCRIPTION

This routine allocates a block of memory for an array that contains *elemNum* elements of size *elemSize* from the shared memory system partition. The return value is the local address of the allocated shared memory block.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the block, or **NULL** if the memory cannot be allocated.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemFindMax( )

NAME **smMemFindMax( )** – find largest free block in shared memory system partition (VxMP Option)

SYNOPSIS

```
int smMemFindMax (void)
```

DESCRIPTION This routine searches for the largest block in the shared memory system partition free list and returns its size.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The size (in bytes) of the largest available block, or **ERROR** if the attempt to access the partition fails.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

smMemFree( )

NAME **smMemFree( )** – free a shared memory system partition block of memory (VxMP Option)

SYNOPSIS

```
STATUS smMemFree
    (
    void * ptr  /* pointer to block of memory to be freed */
    )
```

DESCRIPTION

This routine takes a block of memory previously allocated with **smMemMalloc( )** or **smMemCalloc( )** and returns it to the free shared memory system pool.

It is an error to free a block of memory that was not previously allocated.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the block is invalid.

ERRNO

S_memLib_BLOCK_ERROR
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib, **smMemMalloc( )**, **smMemCalloc( )**

smMemMalloc( )

NAME

smMemMalloc( ) – allocate block of memory from shared memory system partition (VxMP Option)

SYNOPSIS

```
void * smMemMalloc
    (
    unsigned nBytes  /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a block of memory from the shared memory system partition whose size is equal to or greater than *nBytes*. The return value is the local address of the allocated shared memory block.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

A pointer to the block, or **NULL** if the memory cannot be allocated.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib

smMemOptionsSet( )

NAME

smMemOptionsSet( ) – set debug options for shared memory system partition (VxMP Option)

SYNOPSIS

```
STATUS smMemOptionsSet
    (
    unsigned options  /* options for system partition */
    )
```

DESCRIPTION

This routine sets the debug options for the shared system memory partition. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed or reallocated. In both cases, the following options can be selected for actions to be taken when an 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. These options are discussed in detail in the library manual entry for **smMemLib**.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK or **ERROR**.

ERRNO

S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib

smMemRealloc( )

NAME

smMemRealloc( ) – reallocate block of memory from shared memory system partition (VxMP Option)

SYNOPSIS

```
void * smMemRealloc
    (
    void *   pBlock,  /* block to be reallocated */
    unsigned newSize  /* new block size */
    )
```

DESCRIPTION

This routine changes the size of a specified block and returns a pointer to the new block of shared memory. The contents that fit inside the new size (or old size, if smaller) remain unchanged. The return value is the local address of the reallocated shared memory block.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the new block of memory, or **NULL** if the reallocation cannot be completed.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_memLib_BLOCK_ERROR
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemShow( )

NAME **smMemShow( )** – show the shared memory system partition blocks and statistics (VxMP Option)

SYNOPSIS

```
void smMemShow
    (
    int type  /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION This routine displays the total amount of free space in the shared memory system partition, including the number of blocks, the average block size, and the maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

If *type* is 1, it displays a list of all the blocks in the free list of the shared memory system partition.

WARNING This routine locks access to the shared memory system partition while displaying the information. This can compromise the access time to the partition from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smMemShow 1

FREE LIST:
  num    addr       size
  --- ---------- ----------
    1   0x4ffef0        264
    2   0x4fef18       1700

SUMMARY:
    status        bytes    blocks   ave block  max block
--------------- --------- -------- ---------- ----------
        current
           free      1964        2        982       1700
          alloc      2356        1       2356          -
     cumulative
          alloc      2620         2       1310          -
value = 0 = 0x0
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **smMemShow**, **windsh**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*

smNameAdd( )

NAME **smNameAdd( )** – add a name to the shared memory name database (VxMP Option)

SYNOPSIS

```
STATUS smNameAdd
    (
    char * name,   /* name string to enter in database */
    void * value,  /* value associated with name */
    int    type    /* type associated with name */
    )
```

DESCRIPTION This routine adds a name of specified object type and value to the shared memory objects name database.

The *name* parameter is an arbitrary null-terminated string with a maximum of 20 characters, including EOS.

By convention, *type* values of less than 0x1000 are reserved by VxWorks; all other values are user definable. The following types are predefined in **smNameLib.h** :

Name	Value	Type
T_SM_SEM_B	= 0	shared binary semaphore
T_SM_SEM_C	= 1	shared counting semaphore
T_SM_MSG_Q	= 2	shared message queue
T_SM_PART_ID	= 3	shared memory Partition
T_SM_BLOCK	= 4	shared memory allocated block

A name can be entered only once in the database, but there can be more than one name associated with an object ID.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if there is insufficient memory for *name* to be allocated, if *name* is already in the database, or if the database is already full.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_ALREADY_EXIST
S_smNameLib_DATABASE_FULL
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameFind( )

NAME

smNameFind( ) – look up a shared memory object by name (VxMP Option)

SYNOPSIS

```
STATUS smNameFind
    (
    char *  name,      /* name to search for */
    void ** pValue,    /* pointer where to return value */
    int  *  pType,     /* pointer where to return object type */
    int     waitType   /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION

This routine searches the shared memory objects name database for an object matching a specified *name*. If the object is found, its value and type are copied to the addresses pointed to by *pValue* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
The call returns immediately, even if *name* is not in the database.

WAIT_FOREVER (-1)
The call returns only when *name* is available in the database. If *name* is not already in, the database is scanned periodically as the routine waits for *name* to be entered.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the object is not found, if *name* is too long, or the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smNameLib**, **smNameShow**

smNameFindByValue( )

NAME **smNameFindByValue( )** – look up a shared memory object by value (VxMP Option)

SYNOPSIS

```
STATUS smNameFindByValue
    (
    void * value,    /* value to search for */
    char * name,     /* pointer where to return name */
    int  * pType,    /* pointer where to return object type */
    int    waitType  /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION This routine searches the shared memory name database for an object matching a specified value. If the object is found, its name and type are copied to the addresses pointed to by *name* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
The call returns immediately, even if the object value is not in the database.

WAIT_FOREVER (-1)
The call returns only when the object value is available in the database.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if *value* is not found or if the wait type is invalid.

ERRNO **S_smNameLib_NOT_INITIALIZED**
S_smNameLib_VALUE_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smNameLib**, **smNameShow**

smNameRemove( )

NAME

smNameRemove( ) – remove an object from the shared memory objects name database (VxMP Option)

SYNOPSIS

```
STATUS smNameRemove
    (
    char * name  /* name of object to remove */
    )
```

DESCRIPTION

This routine removes an object called *name* from the shared memory objects name database.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the object name is not in the database or if *name* is too long.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameShow( )

NAME

smNameShow( ) – show the contents of the shared memory objects name database (VxMP Option)

SYNOPSIS

```
STATUS smNameShow
    (
    int level  /* information level */
    )
```

DESCRIPTION

This routine displays the names, values, and types of objects stored in the shared memory objects name database. Predefined types are shown, using their ASCII representations; all other types are printed in hexadecimal.

The *level* parameter defines the level of database information displayed. If *level* is 0, only statistics on the database contents are displayed. If *level* is greater than 0, then both statistics and database contents are displayed.

WARNING

This routine locks access to the shared memory objects name database while displaying its contents. This can compromise the access time to the name database from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smNameShow

Names in Database  Max : 30  Current : 6  Free : 24

-> smNameShow 1

Names in Database  Max : 30  Current : 6  Free : 24

Name                 Value         Type
---------------- ----------- -------------
inputImage        0x802340    SM_MEM_BLOCK
ouputImage        0x806340    SM_MEM_BLOCK
imagePool         0x802001    SM_MEM_PART
imageInSem        0x8e0001    SM_SEM_B
imageOutSem       0x8e0101    SM_SEM_C
actionQ           0x8e0201    SM_MSG_Q
userObject        0x8e0400    0x1b0
```

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the name facility is not initialized.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameShow, **smNameLib**

smObjAttach( )

NAME

smObjAttach( ) – attach the calling CPU to the shared memory objects facility (VxMP Option)

SYNOPSIS

```
STATUS smObjAttach
    (
    SM_OBJ_DESC * pSmObjDesc  /* pointer to shared memory descriptor */
    )
```

DESCRIPTION

This routine "attaches" the calling CPU to the shared memory objects facility. The shared memory area is identified by the shared memory descriptor with an address specified by *pSmObjDesc*. The descriptor must already have been initialized by calling **smObjInit( )**.

This routine is called automatically when the component **INCLUDE_SM_OBJ** is included.

This routine will complete the attach process only if and when the shared memory has been initialized by the master CPU. If the shared memory is not recognized as active within the timeout period (10 minutes), this routine returns **ERROR**.

The **smObjAttach( )** routine connects the shared memory objects handler to the shared memory interrupt. Note that this interrupt may be shared between the shared memory network driver and the shared memory objects facility when both are used at the same time.

WARNING

Once a CPU has attached itself to the shared memory objects facility, it cannot be detached. Since the shared memory network driver and the shared memory objects facility use the same low-level attaching mechanism, a CPU cannot be detached from a shared memory network driver if the CPU also uses shared memory objects.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the shared memory objects facility is not active or the number of CPUs exceeds the maximum.

ERRNO

S_smLib_INVALID_CPU_NUMBER

SEE ALSO

smObjLib, **smObjSetup( )**, **smObjInit( )**

smObjGlobalToLocal( )

NAME

smObjGlobalToLocal( ) – convert a global address to a local address (VxMP Option)

SYNOPSIS

```
void * smObjGlobalToLocal
    (
    void * globalAdrs  /* global address to convert */
    )
```

DESCRIPTION

This routine converts a global shared memory address *globalAdrs* to its corresponding local value. This routine does not verify that *globalAdrs* is really a valid global shared memory address.

All addresses stored in shared memory are global. Any access made to shared memory by the local CPU must be done using local addresses. This routine and **smObjLocalToGlobal( )** are used to convert between these address types.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The local shared memory address pointed to by *globalAdrs*.

ERRNO Not Available

SEE ALSO **smObjLib**, **smObjLocalToGlobal( )**

smObjInit( )

NAME **smObjInit( )** – initialize a shared memory objects descriptor (VxMP Option)

SYNOPSIS

```
void smObjInit
    (
    SM_OBJ_DESC * pSmObjDesc,       /* ptr to shared memory descriptor */
    SM_ANCHOR *   anchorLocalAdrs,  /* shared memory anchor local adrs */
    int           ticksPerBeat,     /* cpu ticks per heartbeat */
    int           smObjMaxTries,    /* max no. of tries to obtain spinLock */
    int           intType,          /* interrupt method */
    int           intArg1,          /* interrupt argument #1 */
    int           intArg2,          /* interrupt argument #2 */
    int           intArg3           /* interrupt argument #3 */
    )
```

DESCRIPTION This routine initializes a shared memory descriptor. The descriptor must already be allocated in the CPU's local memory. Once the descriptor has been initialized by this routine, the CPU may attach itself to the shared memory area by calling **smObjAttach( )**.

This routine is called automatically when the component **INCLUDE_SM_OBJ** is included.

Only the shared memory descriptor itself is modified by this routine. No structures in shared memory are affected.

Parameters:

pSmObjDesc
: The address of the shared memory descriptor to be initialized; this structure must be allocated before **smObjInit( )** is called.

anchorLocalAdrs
: The memory address by which the local CPU may access the shared memory anchor. This address may vary among CPUs in the system because of address offsets (particularly if the anchor is located in one CPU's dual-ported memory).

ticksPerBeat
: Specifies the frequency of the shared memory anchor's heartbeat. The frequency is expressed in terms of how many CPU ticks on the local CPU correspond to one heartbeat period.

smObjMaxTries
Specifies the maximum number of tries to obtain access to an internal mutually exclusive data structure.

intType, *intArg1*, *intArg2*, *intArg3*
Allow a CPU to announce the method by which it is to be notified of shared memory events. See the manual entry for **if_sm** for a discussion about interrupt types and their associated parameters.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **smObjLib**, **smObjSetup( )**, **smObjAttach( )**

smObjLibInit( )

NAME **smObjLibInit( )** – install the shared memory objects facility (VxMP Option)

SYNOPSIS
```
STATUS smObjLibInit (void)
```

DESCRIPTION This routine installs the shared memory objects facility. It is called automatically when the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the shared memory objects facility has already been installed.

ERRNO Not Available

SEE ALSO **smObjLib**

smObjLocalToGlobal( )

NAME **smObjLocalToGlobal( )** – convert a local address to a global address (VxMP Option)

SYNOPSIS

```
void * smObjLocalToGlobal
    (
    void * localAdrs  /* local address to convert */
    )
```

DESCRIPTION This routine converts a local shared memory address *localAdrs* to its corresponding global value. This routine does not verify that *localAdrs* is really a valid local shared memory address.

All addresses stored in shared memory are global. Any access made to shared memory by the local CPU must be done using local addresses. This routine and **smObjGlobalToLocal( )** are used to convert between these address types.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The global shared memory address pointed to by *localAdrs*.

ERRNO Not Available

SEE ALSO **smObjLib**, **smObjGlobalToLocal( )**

smObjSetup( )

NAME **smObjSetup( )** – initialize the shared memory objects facility (VxMP Option)

SYNOPSIS

```
STATUS smObjSetup
    (
    SM_OBJ_PARAMS * smObjParams  /* setup parameters */
    )
```

DESCRIPTION This routine initializes the shared memory objects facility by filling the shared memory header. It must be called only once by the shared memory master CPU. It is called automatically only by the master CPU, when the component **INCLUDE_SM_OBJ** is included.

Any CPU on the system backplane can use the shared memory objects facility; however, the facility must first be initialized on the master CPU. Then before other CPUs are attached to the shared memory area by **smObjAttach( )**, each must initialize its own shared memory objects descriptor using **smObjInit( )**. This mechanism is similar to the one used by the shared memory network driver.

The *smObjParams* parameter is a pointer to a structure containing the values used to describe the shared memory objects setup. This structure is defined as follows in **smObjLib.h**:

```
typedef struct sm_obj_params    /* setup parameters */
    {
    BOOL        allocatedPool;  /* TRUE if shared memory pool is malloced */
    SM_ANCHOR * pAnchor;        /* shared memory anchor                   */
    char *      smObjFreeAdrs;  /* start address of shared memory pool    */
    int         smObjMemSize;   /* memory size reserved for shared memory */
    int         maxCpus;        /* max number of CPUs in the system       */
    int         maxTasks;       /* max number of tasks using smObj        */
    int         maxSems;        /* max number of shared semaphores        */
    int         maxMsgQueues;   /* max number of shared message queues    */
    int         maxMemParts;    /* max number of shared memory partitions */
    int         maxNames;       /* max number of names of shared objects  */
    } SM_OBJ_PARAMS;
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the shared memory pool cannot hold all the requested objects or the number of CPUs exceeds the maximum.

ERRNO **S_smObjLib_TOO_MANY_CPU**
S_smObjLib_SHARED_MEM_TOO_SMALL

SEE ALSO **smObjLib**, **smObjInit( )**, **smObjAttach( )**

smObjShow( )

NAME **smObjShow( )** – display the current status of shared memory objects (VxMP Option)

SYNOPSIS

```
STATUS smObjShow (void)
```

DESCRIPTION This routine displays useful information about the current status of shared memory objects facilities.

WARNING The information returned by this routine is not static and may be obsolete by the time it is examined. This information is generally used for debugging purposes only.

EXAMPLE

```
    -> smObjShow
    Shared Mem Anchor Local Addr: 0x600.
    Shared Mem Hdr Local Addr:    0xb1514.
```

```
Attached CPU :                5
Max Tries to Take Lock:       1

Shared Object Type    Current    Maximum  Available
-------------------- ---------- --------- ----------
Tasks                         1        20         19
Binary Semaphores             8        30         20
Counting Semaphores           2        30         20
Messages Queues               3        10          7
Memory Partitions             1         4          3
Names in Database            16       100         84
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if no shared memory objects are initialized.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **smObjShow**, **smObjLib**

smObjTimeoutLogEnable( )

NAME **smObjTimeoutLogEnable( )** – control logging of failed attempts to take a spin-lock (VxMP Option)

SYNOPSIS

```
void smObjTimeoutLogEnable
    (
    BOOL timeoutLogEnable  /* TRUE to enable, FALSE to disable */
    )
```

DESCRIPTION This routine enables or disables the printing of a message when an attempt to take a shared memory spin-lock fails.

By default, message logging is enabled.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **smObjLib**

smeRegister( )

NAME **smeRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void smeRegister(void)
```

DESCRIPTION This routine registers the SMSC driver with VxBus as a child of the PLB bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **vxbSmscLan9118End**

smpLockDemo( )

NAME **smpLockDemo( )** – smpLockDemo entry point (shell command)

SYNOPSIS

```
STATUS smpLockDemo
    (
    unsigned int secs,           /* The minimum life time of a worker task */
    unsigned int reqNumOfTasks,  /* number of tasks */
    BOOL         setAff          /* do task have affinity */
                                 /* numbOfTask is disgarded if affinity set
*/
    )
```

DESCRIPTION Invoke the smpLockDemo by calling this routine from the kernel shell:

```
-> smpLockDemo <number of secs>, <number of tasks>, <[TRUE, FALSE];
(affinity)
```

The *secs* argument is optional. It represents the number of seconds the demo spends updating the local and global counts for each synchronization mechanism mentioned in the module description. When invoked with no arguments the default is two seconds.

RETURNS **OK** if worker tasks were spawn without failure otherwise returns **ERROR**

ERRNO N/A

SEE ALSO **smpLockDemo**

snprintf( )

NAME

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

SYNOPSIS

```
int snprintf
    (
    char *       buffer,  /* buffer to write to */
    size_t       count,   /* max number of characters to store in buffer */
    const char * fmt,     /* format string */
    ...                   /* optional arguments to format */
    )
```

DESCRIPTION

This routine copies a formatted string to a specified buffer, up to a given number of characters. The formatted string will be null terminated. This routine guarantees never to write beyond the provided buffer regardless of the format specifier or the arguments to be formatted. The *count* argument specifies the maximum number of characters to store in the buffer, including the null terminator.

Its function and syntax are otherwise identical to **printf( )**.

RETURNS

The number of characters copied to *buffer*, not including the **NULL** terminator. Even when the supplied *buffer* is too small to hold the complete formatted string, the return value represents the number of characters that would have been written to *buffer* if *count* was sufficiently large.

ERRNO

Not Available

SEE ALSO

fioBaseLib, **sprintf( )**, **printf( )**, *"International Organization for Standardization, ISO/IEC 9899:1999, "*, *"Programming languages - C: Input/output (***stdio.h***)"*

snsShow( )

NAME

snsShow( ) – show information about services in the SNS directory

SYNOPSIS

```
void snsShow
    (
    const char * servName  /* service name prefix */
    )
```

DESCRIPTION

This routine displays information about the services registered with SNS. *servName* is represented in URL format:

[SNS:]service_name[@scope]

where the parts in brackets, [ ], are optional.

SNS: represent the URL service, i.e. the Socket Name Service. It is the only value 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.

All services whose name begins with the string specified by **service_name** are listed. **service_name** may contain wildcard characters ***** or **?** for name pattern matching, where * denotes matching as many characters as possible, including zero number of character, **?** denotes matching any single character.

If *servName* is **NULL**, or points to a null string, then all services in the SNS directory are listed.

The information displayed for each service listed includes:

- the service name
- the service scope
- socket address family associated with the service
- socket type associated with the service
- socket protocol number associated with the service: 0 represents the dummy value, since the socket address family and type are known.
- socket address associated with the service

Examples,

```
NAME............   SCOPE  FAMILY  .TYPE..  PROTO  .... ADDR
-----------------------------------------------------------
webAdmin           node   LOCAL   SEQPKT   0      /comp/socket/0x4
eventBlog          priv   LOCAL   SEQPKT   0      /comp/socket/0x8
clusterTimeServer  clust  TIPC    SEQPKT   0  *   1.1.5,1086717964
```

RETURNS N/A.

ERRNO Not Available

SEE ALSO **snsShow**, **snsLib**

so( )

NAME

so( ) – single-step, but step over a subroutine

SYNOPSIS

```
STATUS so
    (
    int taskNameOrId  /* task to step; 0 = default */
    )
```

DESCRIPTION

This routine single-steps a task that is stopped at a breakpoint. However, if the next instruction is a branch call to a subroutine, **so( )** executes the subroutine and stops after.

To execute, enter:

```
    -> so [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

RETURNS

OK, or **ERROR** if the debugging package is not installed, the task cannot be found, or the task is not suspended.

ERRNO

N/A

SEE ALSO

dbgLib, **s( )**, **cret( )**, the VxWorks programmer guides, the , *VxWorks Command-Line Tools User's Guide*.

sp( )

NAME

sp( ) – spawn a task with default parameters

SYNOPSIS

```
int sp
    (
    FUNCPTR func,  /* function to call       */
    int     arg1,  /* first of nine args to pass to spawned task */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9
    )
```

DESCRIPTION

This command spawns a specified function as a task with the following defaults. These default priorities may be overriden by updating the specified shell variable:

priority (spTaskPriority):
  100

stack size (spTaskStackSize):
  20,000 bytes

task options (spTaskOptions):
  COPROCS_ALL (execute with all coprocessors support)

task name:
  a name of the form **tN** where N is an integer which increments as new tasks are spawned, e.g., **t1**, **t2**, **t3**, etc.

task ID:
  highest not currently used

The task ID is displayed after the task is spawned.

This command is a short form of the underlying **taskSpawn( )** routine, convenient for spawning tasks in which the default parameters are satisfactory. If the default parameters are unacceptable, **taskSpawn( )** should be called directly.

RETURNS

a task ID, or **ERROR** if the task cannot be spawned.

ERRNO

EINVAL
  the address *func* is **NULL**

taskSpawn( ) errnos.

SEE ALSO

usrLib, **taskLib**, **taskSpawn( )**, the VxWorks programmer guides.

spinLockIsrGive( )

NAME

spinLockIsrGive( ) – release an ISR-callable spinlock

SYNOPSIS

```
void spinLockIsrGive
   (
   spinlockIsr_t *pLock  /* pointer to ISR-callable spinlock */
   )
```

DESCRIPTION

This routine releases the ISR-callable spinlock pointed to by *pLock*. Furthermore, it re-enables interrupts that had been disabled on the local CPU when the lock was acquired using **spinLockIsrTake( )**. Calling this routine under the following circumstances is considered to be an error condition and has undefined behaviour:

- The calling task or ISR is not the one that acquired the spinlock.
- The pLock argument does not point to a properly initialized ISR-callable spinlock.

This function forces a read/write memory barrier before releasing the lock.

If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenarios will cause a ED&R kernel fatal error which may reboot the target depending on ED&R policy in place:

- If a CPU different than the owner of a spinlock attempts to release it

- If a CPU attempts to release a spinlock that was never acquired

RETURNS N/A

ERRNO N/A

SEE ALSO **spinLockLib**, **spinLockIsrGive( )**, **spinLockIsrInit( )**

spinLockIsrHeld( )

NAME **spinLockIsrHeld( )** – is an ISR-callable spinlock held by the current CPU?

SYNOPSIS

```
BOOL spinLockIsrHeld
   (
   spinlockIsr_t *pLock  /* pointer to ISR-callable spinlock */
   )
```

DESCRIPTION This routine returns **TRUE** if the ISR-callable spinlock pointed to by *pLock* is currently held by the calling CPU, or **FALSE** if it is not.

Calling this routine with a *pLock* that points to anything that is not a properly initialized ISR-callable spinlock has undefined behaviour.

RETURNS BOOL

ERRNO N/A

SEE ALSO **spinLockLib**, **spinLockIsrGive( )**, **spinLockIsrTake( )**

spinLockIsrInit( )

NAME

spinLockIsrInit( ) – initialize an ISR-callable spinlock

SYNOPSIS

```
VxWorks Architecture Supplements

void spinLockIsrInit
   (
   spinlockIsr_t *pLock,  /* pointer to ISR-callable spinlock */
   int flags                    /* spinlock attributes */
   )
```

DESCRIPTION

This routine initializes the ISR-callable spinlock pointed to by *pLock*, using the *flags* specified. Currently, no flags are defined; this argument is a placeholder for future enhancements. A spinlock must be initialized before it is used for the first time. A spinlock is built on the ability of a processor to perform an atomic read-modify-write access to memory. Some CPUs may have cache attributes and memory alignment restrictions on the use of these instructions. It is the responsibility of the caller to ensure the memory location where the spinlock is located respects these restrictions, if any.

This routine must not be called from interrupt level.

If **INCLUDE_SPINLOCK_DEBUG** is defined, the following scenarios will cause a ED&R kernel fatal error which may reboot the target depending on the ED&R policy in place:

- If this routine is called from interrupt level

RETURNS

N/A

ERRNO

N/A

SEE ALSO

spinLockLib, **spinLockIsrTake( )**, **spinLockIsrGive( )**, **spinLockTaskInit( )**

spinLockIsrTake( )

NAME

spinLockIsrTake( ) – take an ISR-callable spinlock

SYNOPSIS

```
void spinLockIsrTake
    (
#ifdef SPIN_LOCK_TRACE
   spinlockIsr_t *pLock,  /* pointer to ISR-callable spinlock */
   char          *file,
   int         line
#else
   spinlockIsr_t *pLock   /* pointer to ISR-callable spinlock */
#endif
    )
```

RETURNS: N/A

ERRNO: N/A

SEE ALSO: spinLockIsrGive()

<section>
<heading>DESCRIPTION
<p>
This routine acquires the ISR-callable spinlock pointed to by *pLock*.
If the lock is available at the time of the call, this routine
marks the spinlock as being in use and returns immediately.   If the
spinlock is unavailable, the routine busy-waits for the lock to become
available. Because of this busy-wait characteristic, recursive acquisition
of a spinlock causes a live lock situation where the acquiring task or
ISR busy-waits forever for a spinlock it already holds.

<p>
Acquisition of an ISR-callable spinlock causes interrupts to be masked
on the local CPU until the lock is released using **spinLockIsrGive( )**.
It is therefore recommended that this type of spinlock be held for
a minimal amount of time as it increases interrupt latency on
the local CPU.

<p>
Calling this routine with a *pLock* that points to anything that is not a
properly initialized ISR-callable spinlock has undefined behaviour.

<p>
This routine provides a memory barrier mechanism to prevent memory access
reordering that may be performed by the hardware.

<p>
If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenario will inject a
kernel fatal error message in ED&R and may reboot the target depending on
the policy in place:

<p>
- If this routine is called within the context of a CPU already holding the

  same spinlock. A lock can not be taken recursively.

<p>
- If this routine is called while any another ISR-callable is already held.

  ISR-callable spinlocks can not be nested.

<p>
If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenarios will cause a

```
ED&R kernel fatal error which may reboot the target depending on ED&R policy
in place:


<returns>
<heading>RETURNS
<p>
Not Available


<errno>
<heading>ERRNO
<p>
Not Available


<seealso>
<heading>SEE ALSO
<p>
```
spinLockLib

```
<routinedoc>
<rtnhead>spinLockTaskGive( )
<rtnname>
<heading>NAME
<rtnshort>
```
spinLockTaskGive( ) – release a task-only spinlock

```
<synopsis>
<heading>SYNOPSIS
<code>

void spinLockTaskGive
   (
   spinlockTask_t *pLock  /* pointer to task-only spinlock */
   )
```

DESCRIPTION

This routine releases the task-only spinlock pointed to by *pLock*. Furthermore, it re-enables task pre-emption that had been disabled on the local CPU when the lock was acquired using **spinLockTaskTake( )**. Calling this routine under the following circumstances is considered to be an error condition and has undefined behaviour:

- The calling task is not the one that acquired the spinlock.
- The caller is an ISR.
- The pLock argument does not point to a properly initialized task-only spinlock.

This function forces a read/write memory barrier before releasing the lock.

If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenarios will cause a ED&R kernel fatal error which may reboot the target depending on the ED&R policy in place:

- If this routine is called from interrupt level

- If this routine is called within the context of CPU other than owner of the spinlock

RETURNS N/A

ERRNO N/A

SEE ALSO **spinLockLib**, **spinLockTaskTake( )**, **spinLockTaskInit( )**

spinLockTaskInit( )

NAME **spinLockTaskInit( )** – initialize a task-only spinlock

SYNOPSIS

```
void spinLockTaskInit
    (
    spinlockTask_t *pLock,  /* pointer to task-only spinlock */
    int flags                       /* spinlock attributes */
    )
```

DESCRIPTION This routine initializes the task-only spinlock pointed to by *pLock*, using the *flags* specified. Currently, no flags are defined; this argument is a placeholder for future enhancements. A spinlock must be initialized before it is used for the first time. A spinlock is build on the ability of a processor to perform an atomic read-modify-write access to memory. Some CPUs may have cache attributes and memory alignment restrictions on the use of these instructions. It is the responsibility of the caller to ensure the memory location where the spinlock is located respects these restrictions, if any.

This routine must not be called from interrupt level.

If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenarios will cause a ED&R kernel fatal error which may reboot the target depending on the ED&R policy in place:

- If this routine is called from interrupt level

RETURNS N/A

ERRNO N/A

SEE ALSO **spinLockLib**, **spinLockTaskTake( )**, **spinLockTaskGive( )**, **spinLockIsrInit( )**, VxWorks Architecture Supplements

spinLockTaskTake( )

NAME

spinLockTaskTake( ) – take a task-only spinlock

SYNOPSIS

```
void spinLockTaskTake
   (
#ifdef SPIN_LOCK_TRACE
   spinlockTask_t *pLock, /* pointer to task-only spinlock */
   char           *file,
   int          line
#else
   spinlockTask_t *pLock  /* pointer to task-only spinlock */
#endif
   )
```

DESCRIPTION

This routine acquires the task-only spinlock pointed to by *pLock*. If the lock is available at the time of the call, this routine marks the spinlock as being in use and returns immediately. If the spinlock is unavailable, the routine busy-waits for the lock to become available. Because of this busy-wait characteristic, recursive acquisition of a spinlock causes a live lock situation where the acquiring task busy-waits forever for a spinlock it already holds.

Acquisition of an task-only spinlock causes task pre-emption to be disabled on the local CPU until the lock is released using **spinLockTaskGive( )**. It is therefore recommended that this type of spinlock be held for a minimal amount of time as it prevents scheduling on the local CPU.

Calling this routine under the following circumstances is considered to be an error condition and has undefined behaviour:

- Calling this routine with a pLock that points to anything that is not a properly initialized task-only spinlock.
- The caller is an ISR.

This routine provides a memory barrier mechanism to prevent memory access reordering that may be performed by the hardware.

If **INCLUDE_SPINLOCK_DEBUG** is defined the following scenarios will cause a ED&R kernel fatal error which may reboot the target depending on the ED&R policy in place:

- If this routine is called from interrupt level

- If this routine is called within the context of a CPU already holding the target spinlock

- If any other spinlock (any type) is held when this routine is called

RETURNS

N/A

ERRNO

N/A

SEE ALSO **spinLockLib**, **spinLockTaskGive( )**

sprintf( )

NAME **sprintf( )** – write a formatted string to a buffer (ANSI)

SYNOPSIS

```
int sprintf
    (
    char *       buffer,  /* buffer to write to */
    const char * fmt,     /* format string */
    ...                   /* optional arguments to format */
    )
```

DESCRIPTION This routine copies a formatted string to a specified buffer, which is null-terminated. Its function and syntax are otherwise identical to **printf( )**.

RETURNS The number of characters copied to *buffer*, not including the **NULL** terminator.

ERRNO Not Available

SEE ALSO **fioBaseLib**, **printf( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output* (**stdio.h**)

spy( )

NAME **spy( )** – begin periodic task activity reports

SYNOPSIS

```
void spy
    (
    int freq,        /* reporting freq in sec, 0 = default of 5 */
    int ticksPerSec  /* interrupt clock freq, 0 = default of 100 */
    )
```

DESCRIPTION This routine collects task activity data and periodically runs **spyReport( )**. Data is gathered *ticksPerSec* times per second, and a report is made every *freq* seconds. If *freq* is zero, it defaults to 5 seconds. If *ticksPerSec* is omitted or zero, it defaults to 100.

This routine spawns **spyTask( )** to do the actual reporting.

It is not necessary to call **spyClkStart( )** before running **spy( )**.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **spyLib**, **spyClkStart( )**, **spyTask( )**, the VxWorks programmer guides.

spyClkStart( )

NAME **spyClkStart( )** – start collecting task activity data

SYNOPSIS

```
STATUS spyClkStart
    (
    int intsPerSec  /* timer interrupt freq, 0 = default of 100 */
    )
```

DESCRIPTION This routine begins data collection by enabling the auxiliary clock interrupts at a frequency of *intsPerSec* interrupts per second. If *intsPerSec* is omitted or zero, the frequency will be 100. Data from previous collections is cleared.

RETURNS **OK**, or **ERROR** if the CPU has no auxiliary clock, or if task create and delete hooks cannot be installed.

ERRNO N/A

SEE ALSO **usrLib**, **spyLib**, **sysAuxClkConnect( )**, the VxWorks programmer guides.

spyClkStop( )

NAME **spyClkStop( )** – stop collecting task activity data

SYNOPSIS

```
void spyClkStop (void)
```

DESCRIPTION This routine disables the auxiliary clock interrupts. Data collected remains valid until the next **spyClkStart( )** call.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **spyLib**, **spyClkStart( )**, the VxWorks programmer guides.

spyHelp( )

NAME **spyHelp( )** – display task monitoring help menu

SYNOPSIS

```
void spyHelp (void)
```

DESCRIPTION This routine displays a summary of **spyLib** utilities:

```
spyHelp                        Print this list
spyClkStart [ticksPerSec]      Start task activity monitor running
                                 at ticksPerSec ticks per second
spyClkStop                     Stop collecting data
spyReport                      Prints display of task activity
                                 statistics
spyStop                        Stop collecting data and reports
spy     [freq[,ticksPerSec]]   Start spyClkStart and do a report
                                 every freq seconds

ticksPerSec defaults to 100.  freq defaults to 5 seconds.
```

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **spyLib**, the VxWorks programmer guides.

spyLibInit( )

NAME **spyLibInit( )** – initialize task cpu utilization tool package

SYNOPSIS

```
void spyLibInit (void)
```

DESCRIPTION This routine initializes the task cpu utilization tool package. If the configuration macro **INCLUDE_SPY** is defined, it is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **spyLib**, **usrLib**

spyReport( )

NAME
spyReport( ) – display task activity data

SYNOPSIS
```
void spyReport (void)
```

DESCRIPTION
This routine reports on data gathered at interrupt level for 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. Time is displayed in ticks and as a percentage, and the data is shown since both the last call to **spyClkStart( )** and the last **spyReport( )**. If no interrupts have occurred since the last **spyReport( )**, nothing is displayed.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
usrLib, **spyLib**, **spyClkStart( )**, the VxWorks programmer guides.

spyStop( )

NAME
spyStop( ) – stop spying and reporting

SYNOPSIS
```
void spyStop (void)
```

DESCRIPTION
This routine calls **spyClkStop( )**. Any periodic reporting by **spyTask( )** is terminated.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
usrLib, **spyLib**, **spyClkStop( )**, **spyTask( )**, the VxWorks programmer guides.

spyTask( )

NAME
spyTask( ) – run periodic task activity reports

SYNOPSIS
```
void spyTask
```

```
    (
    int freq  /* reporting frequency, in seconds */
    )
```

DESCRIPTION This routine is spawned as a task by **spy( )** to provide periodic task activity reports. It prints a report, delays for the specified number of seconds, and repeats.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **spyLib**, **spy( )**, the VxWorks programmer guides.

sqrtf( )

NAME **sqrtf( )** – compute a non-negative square root (ANSI)

SYNOPSIS

```
float sqrtf
    (
    float x  /* value to compute the square root of */
    )
```

DESCRIPTION This routine returns the non-negative square root of *x* in single precision.

RETURNS The single-precision square root of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

sr( )

NAME **sr( )** – return the contents of the status register (SH)

SYNOPSIS

```
int sr
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION
This command extracts the contents of the status register from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

RETURNS
The contents of the status register.

ERRNO
Not Available

SEE ALSO
dbgArchLib, the VxWorks programmer guides.

sr( )

NAME
sr( ) – return the contents of control register **sr** (also **gbr**, **vbr**) (SH)

SYNOPSIS
```
int sr
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION
This command extracts the contents of register sr from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all control registers (**gbr**, **vbr**): **gbr( )**, **vbr( )**.

RETURNS
The contents of register sr (or the requested control register).

ERRNO
Not Available

SEE ALSO
dbgArchLib, the VxWorks programmer guides.

sscanf( )

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

SYNOPSIS
```
int sscanf
    (
    const char * str,  /* string to scan */
    const char * fmt,  /* format string */
    ...                /* optional arguments to format string */
    )
```

DESCRIPTION

This routine reads characters from the string *str*, interprets them according to format specifications in the string *fmt*, which specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither **%** nor a white-space character); or a conversion specification. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

- An optional assignment-suppressing character *****.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional **h**, **l** (ell) or **ll** (ell-ell) indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** should be preceded by **h** if the corresponding argument is a pointer to `short int' rather than a pointer to **int**, or by **l** if it is a pointer to **long int**, or by **ll** if it is a pointer to **long long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**, or by **ll** if it is a pointer to `unsigned long long int'. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float**. If a **h**, **l** or **ll** appears with any other conversion specifier, the behavior is undefined.
- WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** * argument. However, the current release of the VxWorks libraries does not support **long double** data; using the optional **L** gives unpredictable results.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **sscanf( )** routine executes each directive of the format in turn. If a directive fails, as detailed below, **sscanf( )** returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace( )** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the *fmt* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d
: Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **int**.

i
: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 0 for the *base* argument. The corresponding argument should be a pointer to **int**.

o
: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 8 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

u
: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

x
: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 16 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

e, **f**, **g**
Match an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod( )** function. The corresponding argument should be a pointer to **float**.

s
Matches a sequence of non-white-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[
Matches a non-empty sequence of characters from a set of expected characters (the **scanset**). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent character in the format string, up to and including the matching right bracket (**]**). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (**^**) in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with "[]" or "[^]", the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

c
Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p
Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the **fprintf( )** function. The corresponding argument should be a pointer to a pointer to **void**. VxWorks defines its pointer input field to be consistent with pointers written by the **fprintf( )** function ("0x" hexadecimal notation). If the input item is a value converted earlier during the same program execution, the pointer that results should compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n
No input is consumed. The corresponding argument should be a pointer to **int** into which the number of characters read from the input stream so far by this call to **sscanf( )** is written. Execution of a %n directive does not increment the assignment count returned when **sscanf( )** completes execution.

%
Matches a single **%**; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as **e**, **g**, and **x**, respectively.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or **EOF** if an input failure occurs before any conversion.

ERRNO Not Available

SEE ALSO **fioLib**, **fscanf( )**, **scanf( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output (***stdio.h***)*

ssiDbInit( )

NAME **ssiDbInit( )** – Initialize SSI database.

SYNOPSIS

```
STATUS ssiDbInit
    (
    ssiCompRegInfo_t * pComps
    )
```

DESCRIPTION This routine initializes the SSI database. If **pComps** is not **NULL**, it registers all the components in **pComps** to its database. If the information provided by **pComps** is complete, the SSI dependency tree is also genrated.

pComps points to component registration table, which is used to generate the dependency tree. The component dependency table is considered complete if each of the components in the table have either no dependency or if they have one or many, then all the dependency components are also in the table.

RETURNS **ERROR** if event cannot be sent for one of the following reason:
 - specified descriptor is not valid

- descriptor not specified and could not establish connection
- **cnsWrite( )** fails

ERRNO N/A

SEE ALSO **ssiDb**

ssiShow( )

NAME **ssiShow( )** – Display SSI information

SYNOPSIS

```
STATUS ssiShow
    (
    char * pArgs
    )
```

DESCRIPTION This routine displays information of a specified or all component(s) registered with SSI. The routine can be called with **NULL** argument or with an empty string, in which case all SSI groups and components are displayed.

ERRNO N/A

RETURNS **OK**

ERRNO Not Available

SEE ALSO **ssiDb**

ssmCompInfoGet( )

NAME **ssmCompInfoGet( )** – Get component information.

SYNOPSIS

```
STATUS ssmCompInfoGet
    (
    char *           pName,
    ssmCompInfo_t ** ppInfo
    )
```

DESCRIPTION This routine retrieves the information of a component specified by **pName**.

ARGUMENTS **pName** - specifies the component.

ppInfo - points to a pointer to the component information structure.

RETURNS **ERROR** if the component cannot be found, **OK** otherwise.

ERRNO N/A

SEE ALSO **ssiDb**

ssmCompRegister( )

NAME **ssmCompRegister( )** – Register a component with SSI Manager.

SYNOPSIS

```
STATUS ssmCompRegister
    (
    char *                   pName,
    char *                   pDependencyList,
    SSM_COMP_LAUNCH_FUNCPTR launchFunc,
    char *                   pArgs,
    char *                   pOpts,
    ssmCompInfo_t **         ppComp
    )
```

DESCRIPTION This routine registers a component with the SSI manager.

To participate in the SSI process, components have to be registered with the SSI Manager.

ARGUMENTS **pCompName** is a **NULL** terminated ASCII string that uniquely identifies the component in the system. The string has to be at least 2 byte long (including the terminator) and can be up to **SSM_NAME_LEN** long.

pDependencyList is a list of zero or more names of components that the component requires to wait on before being started. A comma (,) separates each name from another.

launchFunc is a pointer to a function to launch or initialize component. Depending upon whether the component is SSI-aware or not, this routine either starts the component to interact with the CSM or it actually starts the component.

The following is the optional usage of **launchFunc**.
o For an SSI-aware component, **launchFunc** can be **NULL** if the component does not expect to be automatically launched by SSM. If **launchFunc** is provided, SSM launches the component, which will then wait until it receives **CSM_EVENT_INIT** before starting actual execution.

o For a non-compliant component, **launchFunc** does the actual component initialization. There are two ways this can be implemented:
- If the SSM wrapper function is to be used (default for non-compliant comps), the routine is passed to the wrapper function to be called later when the CSM allows the component to initialize.

- If the component opts not to use the wrapper routine, SSM calls the routine directly when the CSM decides the component can be started.

The following table summarizes this usage scenario:
/ts

SSI Aware	USE SSI Wrapper	Init Routine	Init Routine Usage
FALSE	**FALSE**	Required	SSM calls routine to initialize the component.
FALSE	**TRUE**	Reuired	Wrapper task calls routine to initialize the component.
TRUE	**FALSE** | **TRUE**	**NULL**	Component to be launched by some other means.
TRUE	**FALSE** | **TRUE**	Defined	SSM calls routine to launch the component.

/te

The routine should return its state after the initialization and should not block even if the "ready" state is not returned. The implication of this is that dependent components may not be able to proceed.

If the component is running in its own thread, it should send the **CSM_EVENT_READY** eventually. Otherwise, it is recommended that the component uses the default SSM wrapper.

For RTPs, the launch routine is the name of the RTP executable.

For kernel components, if the registration data is obtained from a storage medium, the launch routine field can be 0, in which case the process that is parsing the data has to lookup the component's name in a symbol table to get the initialization function address. If the registration data is input dynamically, the caller of the registration routine may have to pass a function pointer. This is especially true for downloadable kernel modules.

The initialization routine has the following function prototype:

```
STATUS compLaunch (csmStatus_t * pStatus);
```

pArgs points to an ASCCII character string. For kernel components, string format varies from component to component. For RTP components, the string format should include the RTP executable path and name and other RTP attributes including initial arguments.

pOpts represents 0 or more comma-separated strings indicating options. The strings could be one of the following:

"nowarpper" implies that the component does not opt to use the default SSM wrapper routine. Unless the "ssmaware" option is specified, the component's launch/initialization routine is directly called when the CSM allows the component to initialize.

"compliant" implies that component is SSI aware, meaning that the component is compliant with the rules of the SSI system. For SSI aware components, the SSM will simply call the component's registered initialization/launch routine when it is time to launch the component. Subsequently, the initialization process proceeds in the component's own thread or in the thread of the SSM.

"multinst" implies that the component can be started multiple times. This option affects the component's state table. By default, the CSM considers it an error if it receives a **CSM_EV_CREATED** event.

RETURNS **ERROR** if the component cannot be registered.

ERRNO N/A

SEE ALSO **ssiDb**

startupScriptFieldSplit( )

NAME **startupScriptFieldSplit( )** – Split the startup script field of the bootline

SYNOPSIS

```
char * startupScriptFieldSplit
    (
    char * field
    )
```

DESCRIPTION This routine splits the startup script field of the bootline at the first occurence 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.

RETURNS Pointer to a string containing the name of a shell startup script
or **NULL**.

ERRNO N/A.

SEE ALSO **usrRtpStartup**, the VxWorks programmer guides.

stat( )

NAME **stat( )** – get file status information using a pathname (POSIX)

SYNOPSIS

```
STATUS stat
    (
    const char * name,                  /* name of file to check */
    struct stat *pStat                  /* pointer to stat structure */
    )
```

DESCRIPTION

This routine obtains various characteristics of a file (or directory). This routine is equivalent to **fstat( )**, except that the *name* of the file is specified, rather than an open file descriptor.

The *pStat* parameter is a pointer to a **stat** structure (defined in **stat.h**). This structure must have already been allocated before this routine is called.

NOTE

When used with **netDrv** devices (FTP or RSH), **stat( )** returns the size of the file and always sets the mode to regular; **stat( )** does not distinguish between files, directories, links, etc.

Upon return, the fields in the **stat** structure are updated to reflect the characteristics of the file.

RETURNS

OK or **ERROR**, from the underlying io commands **open( )**, **ioctl( )**, or **close( )**.

ERRNO

See **open( )**, **ioctl( )**, and **close( )**.

SEE ALSO

dirLib, **fstat( )**, **ls( )**

statfs( )

NAME

statfs( ) – get file status information using a pathname (POSIX)

SYNOPSIS

```
STATUS statfs
    (
    char           *name,                  /* name of file to check */
    struct statfs *pStat                   /* pointer to statfs structure */
    )
```

DESCRIPTION

This routine obtains various characteristics of a file system. This routine is equivalent to **fstatfs( )**, except that the *name* of the file is specified, rather than an open file descriptor.

The *pStat* parameter is a pointer to a **statfs** structure (defined in **stat.h**). This structure must have already been allocated before this routine is called.

Upon return, the fields in the **statfs** structure are updated to reflect the characteristics of the file.

RETURNS

OK or **ERROR**, from the underlying IO commands **open( )**, **ioctl( )**, **close( )**.

ERRNO

EBADF
Bad file descriptor number.

S_ioLib_UNKNOWN_REQUEST (**ENOSYS**)
Device driver does not support the ioctl command.

ELOOP
Circular symbolic link, too many links.

EMFILE
Maximum number of files already open.

S_iosLib_DEVICE_NOT_FOUND (**ENODEV**)
No valid device name found in path.

Other
Other errors reported by device driver.

SEE ALSO **dirLib**, **fstatfs( )**, **ls( )**

strFree( )

NAME **strFree( )** – free shell strings

SYNOPSIS

```
void strFree
    (
    char * string  /* shell string pointer to free, or 0, or -1 */
    )
```

DESCRIPTION This command free strings allocated within the shell. If *string* is **NULL**, all allocated strings are displayed. If *string* is -1, all allocated strings are freed.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, the VxWorks programmer guides.

swab( )

NAME **swab( )** – swap bytes

SYNOPSIS

```
void swab
    (
    char *source,       /* pointer to source buffer      */
    char *destination,  /* pointer to destination buffer */
    int  nbytes         /* number of bytes to exchange   */
    )
```

DESCRIPTION This routine gets the specified number of bytes from *source*, exchanges the adjacent even and odd bytes, and puts them in *destination*. The buffers *source* and *destination* should not overlap.

NOTE: On some CPUs, **swab( )** will cause an exception if the buffers are unaligned. In such cases, use **uswab( )** for unaligned swaps. On ARM family CPUs, **swab( )** may reorder the bytes incorrectly without causing an exception if the buffers are unaligned. Again, use **uswab( )** for unaligned swaps.

The value of *nBytes* must not be odd. Failure to adhere to this may yield incorrect results.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **uswab( )**

symAdd( )

NAME **symAdd( )** – create and add a symbol to a symbol table, including a group number

SYNOPSIS

```
STATUS symAdd
    (
    SYMTAB_ID symTblId,  /* symbol table to add symbol to */
    char      *name,     /* pointer to symbol name string */
    char      *value,    /* symbol address */
    SYM_TYPE  type,      /* symbol type */
    UINT16    group      /* symbol group */
    )
```

DESCRIPTION This routine allocates a symbol with the specified *name*, *value*, *type*, and *group* and adds it to the symbol table specified by the *symTblId* parameter.

The *group* parameter specifies the group number assigned to a module when it is loaded; see the manual entry for **moduleLib**.

RETURNS **OK**, or **ERROR** if the symbol table is invalid there is insufficient memory for the symbol to be allocated, or any other fatal error occurs.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYMBOL_NAME**
+ **S_symLib_NAME_CLASH**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**, **moduleLib**

symByValueAndTypeFind( )

NAME **symByValueAndTypeFind( )** – look up a symbol by value and type

SYNOPSIS

```
STATUS symByValueAndTypeFind
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    UINT       value,     /* value of symbol to find */
    char **    pName,     /* where to return symbol name string */
    int *      pValue,    /* where to put symbol value */
    SYM_TYPE * pType,     /* where to put symbol type */
    SYM_TYPE   sType,     /* symbol type to look for */
    SYM_TYPE   mask       /* bits in <sType> to pay attention to */
    )
```

DESCRIPTION This routine searches a symbol table for a symbol matching both the specified value and the specified type (*value* and *sType*). If there is no matching entry, it returns the table entry with the next lowest value (among entries with matching type). A pointer to the symbol name string (with terminating EOS) is returned in *pName*. The actual value and the type are copied to *pValue* and *pType*. The *mask* parameter can be used to match sub-classes of type.

pName is a pointer to memory allocated by symByValueAndTypeFind; the memory must be freed by the caller after the use of *pName*.

To search the global VxWorks symbol table, specify **sysSymTbl** as the *symTblId* parameter.

RETURNS **OK** or **ERROR** if *symTblId* is invalid, *pName* is **NULL**, or *value* is less than the lowest value in the table.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**, **symFindSymbol( )**

symByValueFind( )

NAME **symByValueFind( )** – look up a symbol by value

SYNOPSIS

```
STATUS symByValueFind
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    UINT       value,     /* value of symbol to find */
    char **    pName,     /* where return symbol name string */
    int  *     pValue,    /* where to put symbol value */
    SYM_TYPE * pType      /* where to put symbol type */
    )
```

DESCRIPTION This routine searches a symbol table for a symbol whose value matches the specified value. If there is no matching entry, it chooses the table entry with the next lowest value. A pointer to the symbol name string (with terminating EOS) is returned in *pName*. The actual value and the type are copied to the memory pointed to by *pValue* and *pType*.

pName is a pointer to memory allocated by symByValueFind, not to an internal copy of the symbol's name; the memory must be freed by the caller after the use of *pName*.

To search the global VxWorks symbol table, specify **sysSymTbl** as the *symTblId* parameter.

RETURNS **OK** or **ERROR** if *symTblId* is invalid, *pName* is **NULL**, or *value* is less than the lowest value in the table.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**, **symByValueAndTypeFind( )**

symEach( )

NAME **symEach( )** – call a routine to examine each entry in a symbol table

SYNOPSIS

```
SYMBOL * symEach
    (
    SYMTAB_ID symTblId,   /* pointer to symbol table */
    FUNCPTR   routine,    /* func to call for each tbl entry */
    int       routineArg  /* arbitrary user-supplied arg */
    )
```

DESCRIPTION

This routine calls a user-supplied routine to examine each entry in the symbol table; it calls the specified routine once for each entry. The routine should be declared as follows:

```
    BOOL routine
        (
        char *  name,   /* symbol/entry name           */
        int             val,    /* symbol/entry value          */
        SYM_TYPE        type,   /* symbol/entry type           */
        int             arg,    /* arbitrary user-supplied arg */
        UINT16        group   /* symbol/entry group number   */
        )
```

The user-supplied routine should return **TRUE** if **symEach( )** is to continue calling it for each entry, or **FALSE** if it is done and **symEach( )** can exit.

RETURNS

A pointer to the last symbol reached, or **NULL** if all symbols are reached or there is an error.

ERRNO

Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO

symLib

symFindByName( )

NAME

symFindByName( ) – look up a symbol by name

SYNOPSIS

```
STATUS symFindByName
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    char *     name,      /* symbol name to look for */
    char **    pValue,    /* where to return symbol value */
    SYM_TYPE * pType      /* where to return symbol type */
    )
```

DESCRIPTION

This routine searches a symbol table for a symbol matching the specified name. If a symbol is found, its value and type are copied to the memory pointed to by *pValue* and *pType*.

If multiple symbols have the same name, the routine returns the matching symbol most recently added to the symbol table.

To search the global VxWorks (kernel) symbol table, specify **sysSymTbl** as the *symTblId*.

RETURNS **OK**, or **ERROR** if the symbol table ID is invalid or the symbol cannot be found.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**

symFindByNameAndType( )

NAME **symFindByNameAndType( )** – look up a symbol by name and type

SYNOPSIS

```
STATUS symFindByNameAndType
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    char *     name,      /* symbol name to look for */
    char **    pValue,    /* where to put symbol value */
    SYM_TYPE * pType,     /* where to put symbol type */
    SYM_TYPE   sType,     /* symbol type to look for */
    SYM_TYPE   mask       /* bits in <sType> to pay attention to */
    )
```

DESCRIPTION This routine searches a symbol table for a symbol with matching name and type (*name* and *sType*). If the symbol is found, its value and type are copied to the memory pointed to by the pointers *pValue* and *pType*. The *mask* parameter can be used to match sub-classes of type.

To search the global VxWorks (kernel) symbol table, specify **sysSymTbl** as the *symTblId* parameter.

RETURNS **OK**, or **ERROR** if the symbol table ID is invalid or the symbol is not found.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**

symFindByValue( )

NAME **symFindByValue( )** – look up a symbol by value

SYNOPSIS

```
STATUS symFindByValue
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    UINT       value,     /* value of symbol to find */
    char *     name,      /* where to put symbol name string */
    int *      pValue,    /* where to put symbol value */
    SYM_TYPE * pType      /* where to put symbol type */
    )
```

DESCRIPTION This routine is obsolete. It is replaced by **symByValueFind( )** and will be removed in the next version of VxWorks.

This routine searches a symbol table for a symbol matching a specified value. If there is no matching entry, it chooses the table entry with the next lowest value. The symbol name (with terminating EOS), the actual value, and the type are copied to *name*, *pValue*, and *pType*.

For the *name* buffer, allocate **MAX_SYS_SYM_LEN** + 1 bytes. The value **MAX_SYS_SYM_LEN** is defined in **sysSymTbl.h**. If the name of the symbol is longer than **MAX_SYS_SYM_LEN** bytes, it will be truncated to fit into the buffer. Whether or not the name was truncated, the string returned in the buffer will be null-terminated.

To search the global VxWorks symbol table, specify **sysSymTbl** as the *symTblId* parameter.

RETURNS **OK**, or **ERROR** if *symTblId* is invalid or *value* is less than the lowest value in the table.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**

symFindByValueAndType( )

NAME

symFindByValueAndType( ) – look up a symbol by value and type

SYNOPSIS

```
STATUS symFindByValueAndType
    (
    SYMTAB_ID  symTblId,  /* ID of symbol table to look in */
    UINT       value,     /* value of symbol to find */
    char *     name,      /* where to put symbol name string */
    int *      pValue,    /* where to put symbol value */
    SYM_TYPE * pType,     /* where to put symbol type */
    SYM_TYPE   sType,     /* symbol type to look for */
    SYM_TYPE   mask       /* bits in <sType> to pay attention to */
    )
```

DESCRIPTION

This routine is obsolete. It is replaced by the routine **symByValueAndTypeFind( )** and will be removed in the next version of VxWorks.

This routine searches a symbol table for a symbol matching both the specified value and type (*value* and *sType*). If there is no matching entry, it returns the symbol table entry with the next lowest value. The symbol name (with terminating EOS), the actual value, and the type are copied to the memory pointed to by *name*, *pValue*, and *pType*. The *mask* parameter can be used to match sub-classes of type.

For the *name* buffer, allocate **MAX_SYS_SYM_LEN** + 1 bytes. The value **MAX_SYS_SYM_LEN** is defined in **sysSymTbl.h**. If the name of the symbol is longer than **MAX_SYS_SYM_LEN** bytes, it will be truncated to fit into the buffer. Whether or not the name was truncated, the string returned in the buffer will be null-terminated.

To search the global VxWorks symbol table, specify **sysSymTbl** as the *symTblId* parameter.

RETURNS

OK, or **ERROR** if *symTblId* is invalid or *value* is less than the lowest value in the table.

ERRNO

Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO

symLib

symLibInit( )

NAME
symLibInit( ) – initialize the symbol table library

SYNOPSIS
```
STATUS symLibInit (void)
```

DESCRIPTION
This routine initializes the symbol table library. If the configuration macro **INCLUDE_SYM_TBL** is defined, **symLibInit( )** is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS
OK, or **ERROR** if the library could not be initialized.

ERRNO
Not Available

SEE ALSO
symLib

symRemove( )

NAME
symRemove( ) – remove a symbol from a symbol table

SYNOPSIS
```
STATUS symRemove
    (
    SYMTAB_ID symTblId,  /* symbol tbl to remove symbol from */
    char      *name,     /* name of symbol to remove */
    SYM_TYPE  type       /* type of symbol to remove */
    )
```

DESCRIPTION
This routine removes a symbol with matching name and type from a specified symbol table. The symbol is deallocated if found.

Note that VxWorks symbols in a standalone VxWorks image (where the symbol table is linked in) cannot be removed.

RETURNS
OK, or **ERROR** if the symbol is not found or could not be deallocated.

ERRNO
Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_INVALID_SYM_ID_PTR**
+ **S_symLib_SYMBOL_NOT_FOUND**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**

symShow( )

NAME **symShow( )** – show the symbols of specified symbol table with matching substring

SYNOPSIS

```
STATUS symShow
    (
    SYMTAB_ID pSymTbl,  /* ID of symbol table involved */
    char *    substr    /* substring to match */
    )
```

DESCRIPTION This routine lists all symbols in the specified symbol table whose names contain the string *substr*. If *substr* is an empty string (""), all symbols in the table will be listed. If *substr* is **NULL** then the symbol table structure will be summarized

EXAMPLES The system symbol table ID is stored in the global variable *sysSymTbl*.

Look for a symbol containing the "vxWorks" substring (C shell):

```
-> symShow (sysSymTbl, "vxWorks")
```

Print out general information from the system symbol table (C shell):

```
-> symShow (sysSymTbl, 0)
```

Print all symbols contained in the system symbol table (C shell):

```
-> symShow (sysSymTbl,"")
```

RETURNS **OK**, or **ERROR** if the symbol table ID is invalid

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**

For a complete description of the errnos, see the reference documentation for **symShow**.

SEE ALSO **symShow**, **symLib**, **symEach( )**

symShowInit( )

NAME **symShowInit( )** – initialize symbol table show routine

SYNOPSIS

```
void symShowInit (void)
```

DESCRIPTION This routine links the symbol table show facility into the VxWorks system. It is called automatically when the symbol table show facility is configured into VxWorks by including the **INCLUDE_SYM_TBL_SHOW** component.

RETURNS N/A

ERRNO Not Available

SEE ALSO **symShow**

symTblCreate( )

NAME **symTblCreate( )** – create a symbol table

SYNOPSIS

```
SYMTAB_ID symTblCreate
    (
    int     hashSizeLog2,  /* size of hash table as a power of 2 */
    BOOL    sameNameOk,    /* allow 2 symbols of same name & type */
    PART_ID symPartId      /* memory part ID for symbol allocation */
    )
```

DESCRIPTION This routine creates and initializes a symbol table with a hash table of a specified size. The size of the hash table is specified as a power of two. For example, if *hashSizeLog2* is 6, a 64-entry hash table is created.

If the *sameNameOk* parameter is **FALSE**, attempting to add a symbol with the same name and type as an already-existing symbol in the symbol table will result in an error. This behavior cannot be changed once the symbol table has been created.

Memory for storing symbols as they are added to the symbol table will be allocated from the memory partition *symPartId*. Note: the ID of the system memory partition is stored in the global variable **memSysPartId**, which is declared in **memLib.h**.

RETURNS Symbol table ID, or **NULL** if sufficient memory is not available or another fatal error occurred.

ERRNO Not Available

SEE ALSO **symLib**

symTblDelete( )

NAME **symTblDelete( )** – delete a symbol table

SYNOPSIS

```
STATUS symTblDelete
    (
    SYMTAB_ID symTblId  /* ID of symbol table to delete */
    )
```

DESCRIPTION This routine deletes a specified symbol table. It deallocates all associated memory, including the hash table, and marks the table as invalid.

An attempt to delete a table that still contains symbols will return **ERROR**. Successful deletion includes the deletion of the internal hash table and the deallocation of memory associated with the table. The table is marked invalid to prohibit any future references.

RETURNS **OK**, or **ERROR** if the symbol table ID is invalid or if there was a problem.

ERRNO Possible errnos set by this routine include:

+ **S_symLib_INVALID_SYMTAB_ID**
+ **S_symLib_TABLE_NOT_EMPTY**

For a complete description of the errnos, see the reference documentation for **symLib**.

SEE ALSO **symLib**

sysAuxClkConnect( )

NAME **sysAuxClkConnect( )** – connect a routine to the auxiliary clock interrupt

SYNOPSIS

```
STATUS sysAuxClkConnect
    (
    FUNCPTR routine,  /* routine called at each aux clock interrupt    */
    int     arg       /* argument to auxiliary clock interrupt routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each auxiliary clock interrupt. It does not enable auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the routine cannot be connected to the interrupt.

ERRNO Not Available

SEE ALSO **sysLib**, **intConnect( )**, **sysAuxClkEnable( )**, and BSP-specific reference pages for this routine.

sysAuxClkDisable( )

NAME **sysAuxClkDisable( )** – turn off auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkDisable (void)
```

DESCRIPTION This routine disables auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, **sysAuxClkEnable( )**, and BSP-specific reference pages for this routine.

sysAuxClkEnable( )

NAME **sysAuxClkEnable( )** – turn on auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkEnable (void)
```

DESCRIPTION This routine enables auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, **sysAuxClkConnect( )**, **sysAuxClkDisable( )**, **sysAuxClkRateSet( )**, and BSP-specific reference pages for this routine.

sysAuxClkRateGet( )

NAME **sysAuxClkRateGet( )** – get the auxiliary clock rate

SYNOPSIS

```
int sysAuxClkRateGet (void)
```

DESCRIPTION This routine returns the interrupt rate of the auxiliary clock.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the auxiliary clock.

ERRNO Not Available

SEE ALSO **sysLib**, **sysAuxClkEnable( )**, **sysAuxClkRateSet( )**, and BSP-specific reference pages for this routine.

sysAuxClkRateSet( )

NAME **sysAuxClkRateSet( )** – set the auxiliary clock rate

SYNOPSIS

```
STATUS sysAuxClkRateSet
    (
    int ticksPerSecond  /* number of clock interrupts per second */
    )
```

DESCRIPTION This routine sets the interrupt rate of the auxiliary clock. It does not enable auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the tick rate is invalid or the timer cannot be set.

ERRNO Not Available

SEE ALSO **sysLib**, **sysAuxClkEnable( )**, **sysAuxClkRateGet( )**, and BSP-specific reference pages for this routine.

sysBspRev( )

NAME **sysBspRev( )** – return the BSP version and revision number

SYNOPSIS

```
char * sysBspRev (void)
```

DESCRIPTION This routine returns a pointer to a BSP version and revision number, for example, 1.0/1. **BSP_REV** is concatenated to **BSP_VERSION** and returned.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to the BSP version/revision string.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysBusIntAck( )

NAME **sysBusIntAck( )** – acknowledge a bus interrupt

SYNOPSIS

```
int sysBusIntAck
```

```
    (
    int intLevel  /* interrupt level to acknowledge */
    )
```

DESCRIPTION This routine acknowledges a specified VMEbus interrupt level.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **NULL**.

ERRNO Not Available

SEE ALSO **sysLib**, **sysBusIntGen( )**, and BSP-specific reference pages for this routine.

sysBusIntGen( )

NAME **sysBusIntGen( )** – generate a bus interrupt

SYNOPSIS

```
STATUS sysBusIntGen
    (
    int intLevel,  /* bus interrupt level to generate  */
    int vector     /* interrupt vector to generate (0-255)     */
    )
```

DESCRIPTION This routine generates a bus interrupt for a specified level with a specified vector.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if *intLevel* is out of range or the board cannot generate a bus interrupt.

ERRNO Not Available

SEE ALSO **sysLib**, **sysBusIntAck( )**, and BSP-specific reference pages for this routine.

sysBusTas( )

NAME

sysBusTas( ) – test and set a location across the bus

SYNOPSIS

```
BOOL sysBusTas
    (
    char * adrs  /* address to be tested and set */
    )
```

DESCRIPTION

This routine performs a test-and-set instruction across the backplane.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE

This routine is equivalent to **vxTas( )**.

RETURNS

TRUE if the value had not been set but is now, or **FALSE** if the value was set already.

ERRNO

Not Available

SEE ALSO

sysLib, **vxTas( )**, and BSP-specific reference pages for this routine.

sysBusToLocalAdrs( )

NAME

sysBusToLocalAdrs( ) – convert a bus address to a local address

SYNOPSIS

```
STATUS sysBusToLocalAdrs
    (
    int     adrsSpace,  /* bus address space in which busAdrs resides */
    char *  busAdrs,    /* bus address to convert                     */
    char ** pLocalAdrs  /* where to return local address              */
    )
```

DESCRIPTION

This routine gets the local address that accesses a specified bus memory address.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if the address space is unknown or the mapping is not possible.

ERRNO
Not Available

SEE ALSO
sysLib, **sysLocalToBusAdrs( )**, and BSP-specific reference pages for this routine.

sysClkConnect( )

NAME
sysClkConnect( ) – connect a routine to the system clock interrupt

SYNOPSIS
```
STATUS sysClkConnect
    (
    FUNCPTR routine,  /* routine called at each system clock interrupt */
    int     arg       /* argument with which to call routine           */
    )
```

DESCRIPTION
This routine specifies the interrupt service routine to be called at each clock interrupt. Normally, it is called from **usrRoot( )** in **usrConfig.c** to connect **usrClock( )** to the system clock interrupt.

NOTE
This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURN
OK, or **ERROR** if the routine cannot be connected to the interrupt.

RETURNS
Not Available

ERRNO
Not Available

SEE ALSO
sysLib, **intConnect( )**, **usrClock( )**, **sysClkEnable( )**, and BSP-specific reference pages for this routine.

sysClkDisable( )

NAME
sysClkDisable( ) – turn off system clock interrupts

SYNOPSIS
```
void sysClkDisable (void)
```

DESCRIPTION
This routine disables system clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, **sysClkEnable( )**, and BSP-specific reference pages for this routine.

sysClkEnable( )

NAME **sysClkEnable( )** – turn on system clock interrupts

SYNOPSIS

```
void sysClkEnable (void)
```

DESCRIPTION This routine enables system clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, **sysClkConnect( )**, **sysClkDisable( )**, **sysClkRateSet( )**, and BSP-specific reference pages for this routine.

sysClkRateGet( )

NAME **sysClkRateGet( )** – get the system clock rate

SYNOPSIS

```
int sysClkRateGet (void)
```

DESCRIPTION This routine returns the system clock rate.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the system clock.

ERRNO Not Available

SEE ALSO **sysLib**, **sysClkEnable( )**, **sysClkRateSet( )**, and BSP-specific reference pages for this routine.

sysClkRateSet( )

NAME **sysClkRateSet( )** – set the system clock rate

SYNOPSIS

```
STATUS sysClkRateSet
    (
    int ticksPerSecond  /* number of clock interrupts per second */
    )
```

DESCRIPTION This routine sets the interrupt rate of the system clock. It is called by **usrRoot( )** in **usrConfig.c**.

There may be interactions between this routine and the POSIX **clockLib** routines. Refer to the **clockLib** reference entry.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the tick rate is invalid or the timer cannot be set.

ERRNO Not Available

SEE ALSO **sysLib**, **sysClkEnable( )**, **sysClkRateGet( )**, **clockLib**, and BSP-specific reference pages for this routine.

sysHwInit( )

NAME **sysHwInit( )** – initialize the system hardware

SYNOPSIS

```
void sysHwInit (void)
```

DESCRIPTION This routine initializes various features of the board. It is called from **usrInit( )** in **usrConfig.c**.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE This routine should not be called directly by the user application.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysIntDisable( )

NAME **sysIntDisable( )** – disable a bus interrupt level

SYNOPSIS

```
STATUS sysIntDisable
    (
    int intLevel  /* interrupt level to disable */
    )
```

DESCRIPTION This routine disables a specified bus interrupt level.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if *intLevel* is out of range.

ERRNO Not Available

SEE ALSO **sysLib**, **sysIntEnable( )**, and BSP-specific reference pages for this routine.

sysIntEnable( )

NAME

sysIntEnable( ) – enable a bus interrupt level

SYNOPSIS

```
STATUS sysIntEnable
    (
    int intLevel  /* interrupt level to enable (1-7) */
    )
```

DESCRIPTION

This routine enables a specified bus interrupt level.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if *intLevel* is out of range.

ERRNO

Not Available

SEE ALSO

sysLib, **sysIntDisable( )**, and BSP-specific reference pages for this routine.

sysLocalToBusAdrs( )

NAME

sysLocalToBusAdrs( ) – convert a local address to a bus address

SYNOPSIS

```
STATUS sysLocalToBusAdrs
    (
    int     adrsSpace,  /* bus address space in which busAdrs resides */
    char *  localAdrs,  /* local address to convert                   */
    char ** pBusAdrs    /* where to return bus address                */
    )
```

DESCRIPTION

This routine gets the bus address that accesses a specified local memory address.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if the address space is unknown or not mapped.

ERRNO

Not Available

SEE ALSO **sysLib**, **sysBusToLocalAdrs( )**, and BSP-specific reference pages for this routine.

sysMailboxConnect( )

NAME **sysMailboxConnect( )** – connect a routine to the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxConnect
    (
    FUNCPTR routine,  /* routine called at each mailbox interrupt */
    int     arg       /* argument with which to call routine      */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each mailbox interrupt.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the routine cannot be connected to the interrupt.

ERRNO Not Available

SEE ALSO **sysLib**, **intConnect( )**, **sysMailboxEnable( )**, and BSP-specific reference pages for this routine.

sysMailboxEnable( )

NAME **sysMailboxEnable( )** – enable the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxEnable
    (
    char * mailboxAdrs  /* address of mailbox (ignored) */
    )
```

DESCRIPTION This routine enables the mailbox interrupt.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **sysLib**, **sysMailboxConnect( )**, and BSP-specific reference pages for this routine.

sysMemTop( )

NAME **sysMemTop( )** – get the address of the top of logical memory

SYNOPSIS

```
char * sysMemTop (void)
```

DESCRIPTION This routine returns the address of the top of memory.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The address of the top of memory.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysModel( )

NAME **sysModel( )** – return the model name of the CPU board

SYNOPSIS

```
char * sysModel (void)
```

DESCRIPTION This routine returns the model name of the CPU board.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to a string containing the board name.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysNanoDelay( )

NAME **sysNanoDelay( )** – delay for specified number of nanoseconds

SYNOPSIS

```
void sysNanoDelay
    (
    UINT32 nanoseconds  /* nanoseconds to delay */
    )
```

DESCRIPTION This is an optional API for BSPs to provide. Some, but not all, drivers do require the BSP to implement this function.

When implemented, this function implements a spin loop type delay for at least the specified number of nanoseconds. This is not a task delay, control of the processor is not given up to another task. The actual delay must be equal to or greater than the requested number of nanoseconds.

The purpose of this function is to provide a reasonably accurate time delay of very short duration. It should not be used for any delays that are much greater than two system clock ticks in length. For delays of a full clock tick, or more, the use of **taskDelay( )** is recommended.

This routine should be implemented as interrupt safe.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysNetMacNVRamAddrGet( )

NAME **sysNetMacNVRamAddrGet( )** – get network MAC address from NVRAM

SYNOPSIS

```
STATUS sysNetMacNVRamAddrGet
```

```
    (
    char *  ifName,
    int     ifUnit,
    UINT8 * ifMacAddr,
    int     ifMacAddrLen
    )
```

DESCRIPTION

This routine gets the current MAC address from the Non Volatile RAM, and store it in the ifMacAddr buffer provided by the caller.

It is not required for the BSP to provide NVRAM to store the MAC address. Also, some interfaces do not allow the MAC address to be set by software. In either of these cases, this routine simply returns **ERROR**.

Given a MAC address m0:m1:m2:m3:m4:m5, the byte order of ifMacAddr is:
  m0 @ ifMacAddr
  m1 @ ifMacAddr + 1
  m2 @ ifMacAddr + 2
  m3 @ ifMacAddr + 3
  m4 @ ifMacAddr + 4
  m5 @ ifMacAddr + 5

RETURNS

OK, if MAC address available, **ERROR** otherwise

ERRNO

Not Available

SEE ALSO

vxbNonVolLib

sysNvRamGet( )

NAME

sysNvRamGet( ) – get the contents of non-volatile RAM

SYNOPSIS

```
STATUS sysNvRamGet
    (
    char * string,  /* where to copy non-volatile RAM    */
    int    strLen,  /* maximum number of bytes to copy   */
    int    offset   /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION This routine copies the contents of non-volatile memory into a specified string. The string will be terminated with an EOS.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if access is outside the non-volatile RAM address range.

ERRNO Not Available

SEE ALSO **sysLib**, **sysNvRamSet( )**, and BSP-specific reference pages for this routine.

sysNvRamSet( )

NAME **sysNvRamSet( )** – write to non-volatile RAM

SYNOPSIS

```
STATUS sysNvRamSet
    (
    char * string,  /* string to be copied into non-volatile RAM */
    int    strLen,  /* maximum number of bytes to copy           */
    int    offset   /* byte offset into non-volatile RAM         */
    )
```

DESCRIPTION This routine copies a specified string into non-volatile RAM.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if access is outside the non-volatile RAM address range.

ERRNO Not Available

SEE ALSO **sysLib**, **sysNvRamGet( )**, and BSP-specific reference pages for this routine.

sysPhysMemTop( )

NAME

sysPhysMemTop( ) – get the address of the top of memory

SYNOPSIS

```
char * sysPhysMemTop (void)
```

DESCRIPTION

This routine returns the address of the first missing byte of memory, which indicates the top of memory.

Normally, the amount of physical memory is specified with the macro **LOCAL_MEM_SIZE**. BSPs that support run-time memory sizing do so only if the macro **LOCAL_MEM_AUTOSIZE** is defined. If not defined, then **LOCAL_MEM_SIZE** is assumed to be, and must be, the true size of physical memory.

NOTE

Do no adjust **LOCAL_MEM_SIZE** to reserve memory for application use. See **sysMemTop( )** for more information on reserving memory.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

The address of the top of physical memory.

ERRNO

Not Available

SEE ALSO

sysLib, **sysMemTop( )**, and BSP-specific reference pages for this routine.

sysProcNumGet( )

NAME

sysProcNumGet( ) – get the processor number

SYNOPSIS

```
int sysProcNumGet (void)
```

DESCRIPTION

This routine returns the processor number for the CPU board, which is set with **sysProcNumSet( )**.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

The processor number for the CPU board.

ERRNO Not Available

SEE ALSO **sysLib**, **sysProcNumSet( )**, and BSP-specific reference pages for this routine.

sysProcNumSet( )

NAME **sysProcNumSet( )** – set the processor number

SYNOPSIS

```
void sysProcNumSet
    (
    int procNum  /* processor number */
    )
```

DESCRIPTION This routine sets the processor number for the CPU board. Processor numbers should be unique on a single backplane.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, **sysProcNumGet( )**, and BSP-specific reference pages for this routine.

sysScsiBusReset( )

NAME **sysScsiBusReset( )** – assert the RST line on the SCSI bus (Western Digital WD33C93 only)

SYNOPSIS

```
void sysScsiBusReset
    (
    FAST WD_33C93_SCSI_CTRL * pSbic  /* ptr to SBIC info */
    )
```

DESCRIPTION This routine asserts the RST line on the SCSI bus, which causes all connected devices to return to a quiescent state.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysScsiConfig( )

NAME **sysScsiConfig( )** – system SCSI configuration

SYNOPSIS

```
STATUS sysScsiConfig (void)
```

DESCRIPTION This is an example SCSI configuration routine.

Most of the code found here is an example of how to declare a SCSI peripheral configuration. You must edit this routine to reflect the actual configuration of your SCSI bus. This example can also be found in **src/config/usrScsi.c**.

If you are just getting started, you can test your hardware configuration by defining **SCSI_AUTO_CONFIG**, which will probe the bus and display all devices found. No device should have the same SCSI bus ID as your VxWorks SCSI port (default = 7), or the same as any other device. Check for proper bus termination.

There are two configuration examples here. They demonstrate configuration of a SCSI hard disk (any type) and an OMTI 3500 floppy disk.

Hard Disk The hard disk is divided into two 32-Mbyte partitions and a third partition with the remainder of the disk.

It is recommended that the first partition (**BLK_DEV**) on a block device be a dosFs device, if the intention is eventually to boot VxWorks from the device. This will simplify the task considerably.

Floppy Disk

The floppy, since it is a removable medium device, is allowed to have only a single partition.

In contrast to the hard disk configuration, the floppy setup in this example is more intricate. Note that the **scsiPhysDevCreate( )** call is issued twice. The first time is merely to get a "handle" to pass to **scsiModeSelect( )**, since the default media type is sometimes inappropriate (in the case of generic SCSI-to-floppy cards). After the hardware is correctly configured, the handle is discarded via **scsiPhysDevDelete( )**, after which the peripheral is correctly configured by a second call to **scsiPhysDevCreate( )**. (Before the **scsiModeSelect( )** call, the configuration information was incorrect.) Note that after the **scsiBlkDevCreate( )** call, the correct values for *sectorsPerTrack* and *nHeads* must be set via **scsiBlkDevInit( )**. This is necessary for IBM PC compatibility.

NOTE

The variable **pSbdFloppy** is global to allow the above calls to be made from the VxWorks shell, for example:

```
    -> dosFsMkfs "/fd0", pSbdFloppy
```

If a disk is new, use **diskFormat( )** to format it.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK or **ERROR**.

ERRNO

Not Available

SEE ALSO

sysLib, and BSP-specific reference pages for this routine.

sysScsiInit( )

NAME

sysScsiInit( ) – initialize an on-board SCSI port

SYNOPSIS

```
STATUS sysScsiInit (void)
```

DESCRIPTION

This routine creates and initializes a SCSI control structure, enabling use of the on-board SCSI port. It also connects the proper interrupt service routine to the desired vector, and enables the interrupt at the desired level.

If SCSI DMA is supported by the board and **INCLUDE_SCSI_DMA** is defined, the DMA is also initialized.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if the control structure cannot be connected, the controller cannot be initialized, or the DMA's interrupt cannot be connected.

ERRNO

Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialChanGet( )

NAME **sysSerialChanGet( )** – get the **SIO_CHAN** device associated with a serial channel

SYNOPSIS

```
SIO_CHAN * sysSerialChanGet
    (
    int channel  /* serial channel */
    )
```

DESCRIPTION This routine gets the **SIO_CHAN** device associated with a specified serial channel.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to the **SIO_CHAN** structure for the channel, or **ERROR** if the channel is invalid.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialHwInit( )

NAME **sysSerialHwInit( )** – initialize the BSP serial devices to a quiesent state

SYNOPSIS

```
void sysSerialHwInit (void)
```

DESCRIPTION This routine initializes the BSP serial device descriptors and puts the devices in a quiesent state. It is called from **sysHwInit( )** with interrupts locked.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialHwInit2( )

NAME **sysSerialHwInit2( )** – connect BSP serial device interrupts

SYNOPSIS

```
void sysSerialHwInit2 (void)
```

DESCRIPTION This routine connects the BSP serial device interrupts. It is called from **sysHwInit2( )**. Serial device interrupts could not be connected in **sysSerialHwInit( )** because the kernel memory allocator was not initialized at that point, and **intConnect( )** calls **malloc( )**.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialReset( )

NAME **sysSerialReset( )** – reset all SIO devices to a quiet state

SYNOPSIS

```
void sysSerialReset (void)
```

DESCRIPTION This routine is called from **sysToMonitor( )** to reset all SIO device and prevent them from generating interrupts or performing DMA cycles.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysToMonitor( )

NAME **sysToMonitor( )** – transfer control to the ROM monitor

SYNOPSIS

```
STATUS sysToMonitor
    (
    int startType  /* parameter passed to ROM to tell it how to boot */
    )
```

DESCRIPTION This routine transfers control to the ROM monitor. Normally, it is called only by **reboot( )**--which services ^X--and by bus errors at interrupt level. However, in some circumstances, the user may wish to introduce a *startType* to enable special boot ROM facilities.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS Does not return.

ERRNO Not Available

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

syscallDispatch( )

NAME **syscallDispatch( )** – dispatch a system call request to its system call handler

SYNOPSIS

```
STATUS syscallDispatch
    (
    SYSCALL_ENTRY_STATE * pState
    )
```

DESCRIPTION This routine is the system call dispatcher. It decodes a system call number into the group and routine numbers, and locates the proper handler function to call for the given system call request. The system call group must have been previously configured at build time, or registered by calling **syscallGroupRegister( )**.

Users can hook into the system call dispatch process by attaching entry and exit hook funcitons. These hooks are installed via functions **syscallEntryHookAdd( )** and **syscallExitHookAdd( )** respectively.

Entry hooks are called after decoding the system call number and verifying that this is a valid system call, but before the handler is called. Entry hooks are passed a pointer to the system call entry state structure as a parameter. Since the state structure contains vital machine information about the trap conditions, it is recommended that its contents not be changed. No explicit barriers prevent such a change from being made. However it is the authors duty to warn you that an ill-considered hook modifying machine state information can crash the system easily. So please do so only if you must, and with the utmost caution. If an entry hook does modify state information, the system call handler function sees the changed values. The state information is architecture-specific. Entry hook functions must return a STATUS value. Any return value other than **OK** causes the dispatcher to return **ERROR** back to user code instead of calling the system call handler function. Thus, entry hooks can prevent an otherwise valid system call from being executed.

Exit hooks are called after the system call handler function returns. They are passed the handler's return value as a parameter. Exit hooks are not expected to return anything.

RETURNS The return value from the system call handler called, or **ERROR**.

ERRNO **ENOSYS**
invalid system call request, or no handler function set.

SEE ALSO **syscallLib**, the VxWorks programmer guides.

syscallEntryHookAdd( )

NAME **syscallEntryHookAdd( )** – add a routine to be called on each system call entry

SYNOPSIS

```
STATUS syscallEntryHookAdd
    (
    SYSCALL_ENTRY_HOOK hook,      /* routine to call upon system call entry
*/
    BOOL               addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called when a system call is made. The hook routine should have the following prototype:

```
STATUS syscallEntryHook
    (
    SYSCALL_ENTRY_STATE * pState     /* system call entry state */
    )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (in other words, it will be the first hook to execute).

System call entry hooks are called from the system call dispatcher after the system call is decoded, but before the handler function is called. Hook functions should return either **OK** or **ERROR**. If the return value from any hook is anything other than **OK**, the system call is aborted and **ERROR** is returned back to the user. Entry hooks can be used to implement rudimentary authentication schemes by rejecting otherwise valid system calls.

RETURNS **OK**, or **ERROR** if the table of hook routines table is full.

ERRNO N/A.

SEE ALSO **syscallHookLib**, **syscallEntryHookDelete( )**

syscallEntryHookDelete( )

NAME **syscallEntryHookDelete( )** – delete a previously added entry hook

SYNOPSIS

```
STATUS syscallEntryHookDelete
    (
    SYSCALL_ENTRY_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of system call entry hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **syscallHookLib**, **syscallPreCreateHookAdd( )**

syscallExitHookAdd( )

NAME **syscallExitHookAdd( )** – add a routine to be called on each system call exit

SYNOPSIS

```
STATUS syscallExitHookAdd
```

```
    (
    SYSCALL_EXIT_HOOK hook,      /* routine to call upon system call exit */
    BOOL              addToHead  /* add routine to head of list? */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called when a system call is about to return back to the user. The hook routine should have the following prototype:

```
    void syscallExitHook
        (
        int returnValue  /* system call return value */
        )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (in other words, it will be the first hook to execute).

System call exit hooks are called from the system call dispatcher before the system call exits back to the user. Exit hooks are not expected to return anything (return values are not checked).

RETURNS

OK, or **ERROR** if the table of hook routines table is full.

ERRNO

N/A.

SEE ALSO

syscallHookLib, **syscallExitHookDelete( )**

syscallExitHookDelete( )

NAME

syscallExitHookDelete( ) – delete a previously added exit hook

SYNOPSIS

```
STATUS syscallExitHookDelete
    (
    SYSCALL_EXIT_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION

This routine removes a specified hook routine from the list of system call exit hook routines.

RETURNS

OK on success, or **ERROR** if the hook routine was not found.

ERRNO

S_hookLib_HOOK_NOT_FOUND

SEE ALSO

syscallHookLib, **syscallExitHookAdd( )**

syscallGroupRegister( )

NAME

syscallGroupRegister( ) – register a system call group with the SCI

SYNOPSIS

```
STATUS syscallGroupRegister
    (
    int                      groupNum,
    char *                   groupName,
    int                      numRoutines,
    SYSCALL_RTN_TBL_ENTRY *  pRoutineTbl,
    BOOL                     force         /* forcibly overwrite an existing
*/
                                           /* entry helps when debugging */
    )
```

DESCRIPTION

This routine registers a system call group with the System Call Infrastructure. Registration is a must, without which user-level code cannot make any system calls to the group in question.

Users can hook into the registration process by adding a registration hook function using **syscallRegisterHookAdd( )**. Any attached hooks functions will be called prior to actually registering the group in question. The hook functions are passed the same parameters as are passed to this function with the exception of the force parameter. Hook functions are expected to return a STATUS value. Any return value other than **OK** causes this function to return **ERROR** instead of performing the actual registration. Thus, the registration hook can prevent an otherwise valid registration operation from proceeding. This function performs parameter validation prior to calling the registration hooks if any.

RETURNS

OK on success, **ERROR** otherwise.

ERRNO

S_syscallLib_UNKNOWN_GROUP

S_syscallLib_GROUP_EXISTS

S_syscallLib_TOO_MANY_ROUTINES

S_syscallLib_NO_ROUTINES_TBL

SEE ALSO

syscallLib, **syscallRegisterHookAdd( )**, **syscallRegisterHookDelete( )**, the VxWorks programmer guides.

syscallHookShow( )

NAME **syscallHookShow( )** – display all installed system call infrastructure hooks

SYNOPSIS

```
void syscallHookShow (void)
```

DESCRIPTION This routine displays the contents of all three system call infrastructure hook tables - the entry, exit and registration hook tables.

EXAMPLE The following example shows hypothetical system call hook table contents:

```
-> syscallHookShow

System Call Entry Hook Table:

entryHook1
entryHook2
entryHook3

System Call Exit Hook Table:

exitHook

System Call Registration Hook Table:

registrationHookA
registrationHookB
value = 1 = 0x1
->
```

RETURNS N/A

ERRNOS N/A

SEE ALSO **syscallShow**, **rtpHookShow( )**, **hookShow( )**

syscallMonitor( )

NAME **syscallMonitor( )** – monitor system call activity

SYNOPSIS

```
void syscallMonitor
    (
    int    level,
    RTP_ID rtpId
    )
```

DESCRIPTION

This routine enables/disables system call monitoring activity. It behaves a little like the BSD ktrace or Solaris truss utilities. Users can monitor either a single or all RTPs in the system. When monitoring is turned on, all system calls made by the target RTP are displayed on the console with their argument and return values. This helps monitor all system calls made by applications.

syscallMonitor works via the system call hook facility (i.e. **syscallHookLib**) It preserves other hooks previously installed. Enabling this facility more than once has no effect.

The first parameter enables monitoring (level = 1), or disables it (level = 0). The second parameter is the **RTP_ID** of the RTP to monitor system calls for. If it is 0, all RTPs are monitored.

In SMP version of VxWorks, all actions regarding system calls are guaranteed to be logged only once syscallMonitor returns.

RETURNS

N/A.

ERRNO

None.

SEE ALSO

syscallShow, the VxWorks programmer guides.

syscallRegisterHookAdd( )

NAME

syscallRegisterHookAdd( ) – add hook for system call group registration requests

SYNOPSIS

```
STATUS syscallRegisterHookAdd
    (
    SYSCALL_REGISTER_HOOK hook,      /* routine to call upon group
registration */
    BOOL                  addToHead  /* add routine to head of list */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called just before an system call group is registered. The hook routine should have the following prototype:

```
    STATUS syscallRegistrationHook
        (
        int                      groupNum,        /* group number */
        char *                   groupName,       /* group name */
        int                      numRoutines,     /* num routines in group */
        SYSCALL_RTN_TBL_ENTRY ** ppRoutineTbl, /* addr of routine table */
        )
```

The second parameter *addToHead* specifies the order in which the hook is added to the table. If **FALSE**, the hook is appended to the list of hooks already installed. If addToHead is **TRUE**, the new hook is added to the head of the list (in other words, it will be the first hook to execute).

System call registration hooks are called from **syscallGroupRegister( )** before the registration is actually done. Each hook function should return either **OK** or **ERROR**. If the return value from any hook is anything other than **OK**, registration is aborted and **ERROR** is returned from **syscallGroupRegister( )**. Registration hooks can be used to implement rudimentary authentication schemes by rejecting otherwise valid group registration requests.

RETURNS **OK**, or **ERROR** if the table of hook routines table is full.

ERRNO N/A.

SEE ALSO **syscallHookLib**, **syscallRegisterHookDelete( )**

syscallRegisterHookDelete( )

NAME **syscallRegisterHookDelete( )** – delete a previously added registration hook.

SYNOPSIS

```
STATUS syscallRegisterHookDelete
    (
    SYSCALL_REGISTER_HOOK hook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified hook routine from the list of system call registration hook routines.

RETURNS **OK** on success, or **ERROR** if the hook routine was not found.

ERRNO **S_hookLib_HOOK_NOT_FOUND**

SEE ALSO **syscallHookLib**, **syscallRegisterHookAdd( )**

syscallShow( )

NAME **syscallShow( )** – show registered System Call Groups, or a specific group

SYNOPSIS

```
void syscallShow
```

```
    (
    int grp,
    int level
    )
```

DESCRIPTION

This routine shows the registered System Call Groups, as well as details of routines exported by a specific group. When the level parameter is 0, the *grp* parameter is ignored, and all registered groups are shown. The level 0 output (equivalent to **syscallShow** (0,0)) on a target shell looks like this:

```
-> syscallShow
Group Name              GroupNo   NumRtns   Rtn Tbl Addr
--------------------    -------   -------   ------------
STANDARDGroup             8         49       0x00274bc0
VXWORKSGroup              9         51       0x00274ed0
value = 53 = 0x35 = '5'
->
```

When the level parameter is 1, details of the System Call Group *grp* are shown. The level 1 output for one the above group looks like this:

```
-> syscallShow 8,1
System Call Group name: STANDARDGroup
Group Number          : 8

Routines provided     :
Rtn#   Name                       Address     # Arguments
----   ----------------------   ----------   -----------
0      _exit                    0x001f88fc      1
1      creat                    0x001e631c      2
2      open                     0x001e62b8      3
3      close                    0x001e643c      1
4      read                     0x001e6444      3
5      write                    0x001e63d8      3
6      ioctl                    0x001e64a8      3
7      dup                      0x001e66d0      1
8      dup2                     0x001e66d8      2
9      pipe                     0x001e66f0      1
10     remove                   0x001e637c      1
11     select                   0x001e64c4      5
12     socket                   0x001aa5e4      3
13     bind                     0x001aa6e4      3
14     listen                   0x001aa7c4      2
15     accept                   0x001aa85c      3
16     connect                  0x001aa9a0      3
17     sendto                   0x001aac38      6
18     send                     0x001aad58      4
19     sendmsg                  0x001ab718      3
20     recvfrom                 0x001aae34      6
21     recv                     0x001aaf7c      4
22     recvmsg                  0x001ab5c8      3
23     setsockopt               0x001ab064      5
24     getsockopt               0x001ab164      5
25     getsockname              0x001ab278      3
26     getpeername              0x001ab378      3
```

```
27     shutdown                 0x001ab478      2
28     mmap                     0x001ed47c      8
29     munmap                   0x001ed4a4      2
30     mprotect                 0x001ed4bc      3
31     kill                     0x001f8c54      2
32     pause                    0x001f8bdc      0
33     sigpending               0x001f8bcc      1
34     sigprocmask              0x001f8adc      3
35     _sigqueue                0x001f8cb0      3
36     sigsuspend               0x001f8bd4      1
37     sigtimedwait             0x001f8d34      3
38     _sigaction               0x001f8a2c      4
39     _sigreturn               0x001f8d10      0
40     chdir                    0x001e65f4      1
41     _getcwd                  0x001e6650      2
42     symlink                  0x001e6584      2
43     getpid                   0x001f8e64      0
44     getppid                  0x001f8e88      0
45     waitpid                  0x001f8dd8      3
46     sysctl                   0x00164a34      6
47     _schedPxInfoGet          0x001dab00      2
48     sigaltstack              0x001f8af8      2
value = 50 = 0x32 = '2'
->
```

RETURNS N/A.

ERRNO None.

SEE ALSO **syscallShow**, the VxWorks programmer guide.

sysctl( )

NAME **sysctl( )** – get or set the the values of objects in the sysctl tree

SYNOPSIS

```
int sysctl
    (
    int *    pName,    /* Name vector of object in MIB style */
    u_int    nameLen,  /* Number of elements in the name vector */
    void *   pOld,     /* Buffer to place the current value of object */
    size_t * pOldLen,  /* Buffer for the size of current value */
    void *   pNew,     /* Buffer containing a value to set, if needed */
    size_t   newLen    /* Size of the buffer containing new value */
    )
```

DESCRIPTION This routine retrieves system state information and allows the setting of system information, provided that they have appropriate privileges. The information that sysctl returns will be either an integer, string or table. The state description, hold by the *pName* parameter, is in a MIB or Management Information Base style: a vector of integers. The number of elements in the name vector is specified via the *nameLen* parameter.

The information is copied into the buffer specified by *pOld*. The size of the buffer is given by the location specified by *pOldLen* before the call, and that location gives the amount of data copied after a successful call and after a call that returns with the error code **ENOMEM**. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code **ENOMEM**. If the old value is not desired, *pOld* and *pOldLen* should be set to **NULL**.

The size of the available data can be determined by calling **sysctl( )** with a **NULL** parameter for *pOld*. The size of the available data will be returned in the location pointed to by *pOldLen*. For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *pNew* is set to point to a buffer of length *newLen*.

If a new value doesn't need to be set, *pNew* should be set to **NULL**, and *newLen* should be set to 0.

The name vector's elements correspond to a hierarchy of integer values which description can be found in **sys/sysctl.h**. The top level names start with the **CTL_** prefix, for instance **CTL_KERN**. The second level names start with a prefix referring to the top level name they are related to, for instance **KERN_OSTYPE**, etc.

For instance to get the name of the CPU family on a target board:

```
int mib[4];
char cpuFamily[10];
int nameSize = sizeof (cpuFamily);

/* Fill out the MIB-style name vector */

mib[0] = CTL_HW;
mib[1] = HW_PAL;
mib[2] = HW_PAL_CPU;
mib[3] = HW_PAL_CPU_FAMILY;

/* Fetch and print the CPU family name */

if (sysctl (mib, 4, (void *)&cpuFamily, (size_t *)&nameSize,
            NULL, 0) == -1)
    printf ("Failed getting the CPU family name\n");
else
    printf ("CPU family name: %s\n", cpuFamily);
```

RETURNS 0 on success, or -1 if an error occurred.

ERRNO Beside the errnos possibly set by the OID's non-default handler the following errnos may be set:

EPERM
An attempt is made to set a read-only value.

EINVAL
The name vector has less than two or more than **CTL_MAXNAME** elements, or the OID is not a node and has no handler, or the *newLen* size of the *pNew* buffer is too small.

ENOMEM
the *pOldLen* size of the *pOld* buffer is too small for the requested information to be stored in this buffer.

ENOENT
The OID does not exist.

EISDIR
The OID is a node without a handler so no information can be set or retreived.

ENOTDIR
One of the OID numbers in the name vector, except for the last element, does not correspond to a node OID.

SEE ALSO **kern_sysctl**, **sysctlbyname( )**, **sysctlnametomib( )**

sysctl_add_oid( )

NAME **sysctl_add_oid( )** – add a parameter into the sysctl tree during run-time

SYNOPSIS

```
struct sysctl_oid * sysctl_add_oid
    (
    struct sysctl_ctx_list * clist,
    struct sysctl_oid_list * parent,
    int                      number,
    const char *             name,
    int                      kind,
    void *                   arg1,
    int                      arg2,
    int                      (*handler)(SYSCTL_HANDLER_ARGS),
    const char *             fmt,
    const char *             descr
    )
```

DESCRIPTION This routine allows the dynamic addition of a parameter that needs to be accessed via sysctl. To use this API, the following arguments are needed.

clist
This is always set to **NULL** since user-defined contexts are currently not supported.

parent
The node under which this object is to be registered. When connecting to the static node available for user extensions, **usr_ext**, use the **SYSCTL_NODE_CHILDREN(usr_ext)** macro to get the pointer to this static OID. When connecting to a dynamic node use the **SYSCTL_CHILDREN( )** macro with the name of the pointer of type **struct sysctl_oid** representing the parent OID.

number
the OID number that will be assigned to this object. It is highly recommended to use the **OID_AUTO** macro to avoid conflict with already registered OIDs.

name
A user-specified name for this object.

kind
The kind of object this OID represents as well as the access permissions it holds. For instance **CTLTYPE_NODE** | **CTLFLAG_RD**, **CTLTYPE_STRING** | **CTLFLAG_RW**, **CTLTYPE_INT** | **CTLFLAG_RW**, etc. See **sys/sysctl.h** for the full list of the CTLTYPE_... and CTLFLAG_... macros.

arg1
A pointer to any data that the OID should reference, or **NULL**. See the SYSCTL_ADD_... macros below for specific details.

arg2
The size of *arg1* or 0 if *arg1* is **NULL**.

handler
A pointer to the function that will handle read and write requests to this OID. A set of standard handlers are provided that support operations on integers (**sysctl_handle_int( )**), strings (**sysctl_handle_string( )**) and opaque objects (**sysctl_handle_opaque( )**). New handlers can be created (see **SYSCTL_ADD_PROC( )** below).

fmt
A pointer to a string that specifies the format of this object. The string must hold "N" for nodes, "A" for strings, "I" for integers, "IU" for unsigned integers, "L" for longs, "LU" for unsigned longs and "S, <type>" for structures (see **SYSCTL_ADD_STRUCT( )** below for details).

descr
An optional description string.

Utility macros are provided to simplify the creation of OIDs. These macros are:

SYSCTL_ADD_OID( )
Creates a raw OID. This is equivalent to calling **sysctl_add_oid( )**. The parameters are therefore the same as for the **sysctl_add_oid( )** routine:

```
SYSCTL_ADD_OID (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                parent, int number, const char * name, int kind, void * arg1,
                int arg2, int (*handler)(SYSCTL_HANDLER_ARGS), const char *
                fmt, const char * descr);
```

SYSCTL_ADD_NODE( )

Creates an OID of type **CTLTYPE_NODE**. Other OIDs can be added to nodes, as children:

```
SYSCTL_ADD_NODE (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                 parent, int number, const char * name, int access,
                 int (*handler)(SYSCTL_HANDLER_ARGS), const char * descr);
```

The *access* parameter represents a combination of access permission flags (CTLFLAG_ macros).

SYSCTL_ADD_STRING( )

Creates an OID of type **CTLTYPE_STRING** that handles a null-terminated character string:

```
SYSCTL_ADD_STRING (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                   parent, int number, const char * name, int access,
                   char * arg, int len, const char * descr);
```

The *arg* parameter is the address of the string variable and *len* is the maximum length of the string referred to by this OID should it be changed later.

SYSCTL_ADD_INT( )

Creates an OID of type **CTLTYPE_INT** that handles an integer type variable:

```
SYSCTL_ADD_INT (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                parent, int number, const char * name, int access, int * arg,
                int len, const char * descr);
```

The *access* parameter represents a combination of access permission flags (CTLFLAG_ macros). The *arg* parameter is the address of the integer variable and *len* is its length.

SYSCTL_ADD_UINT( )

Creates an OID of type **CTLTYPE_INT** and format "IU" that handles an unsigned integer type variable. The parameters are the same as for **SYSCTL_ADD_INT( )** except for the *arg* parameter which is of type "unsigned int *".

SYSCTL_ADD_LONG( )

Creates an OID of type **CTLTYPE_INT** and format "L" that handles a long integer type variable. The parameters are the same as for **SYSCTL_ADD_INT( )** except for the *arg* parameter which is of type "long *".

SYSCTL_ADD_ULONG( )

Creates an OID of type **CTLTYPE_INT** and format "LU" that handles an unsigned long integer type variable. The parameters are the same as for **SYSCTL_ADD_INT( )** except for the *arg* parameter which is of type "unsigned long *".

SYSCTL_ADD_OPAQUE( )

Creates an OID of type **CTLTYPE_OPAQUE** that handles an unspecified block of data:

```
SYSCTL_ADD_OPAQUE (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                   parent, int number, const char * name, int access,
                   void * arg, int len, const char * format, const char *
                   descr);
```

The *access* parameter represents a combination of access permission flags (CTLFLAG_ macros). The *arg* parameter is the address of the opaque data and *len* is its length.

SYSCTL_ADD_STRUCT( )
Creates an OID that handles a structure type variable. The type of the OID will be **CTLTYPE_OPAQUE** and its format will indicates the type name "S, <type>":

```
SYSCTL_ADD_STRUCT (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                   parent, int number, const char * name, int access,
                   void * arg, TYPE, const char * descr);
```

The *access* parameter represents a combination of access permission flags (CTLFLAG_ macros). The *arg* parameter is the address of the stucture variable data and *TYPE* is its type name (without the struct keyword) used in the format string.

SYSCTL_ADD_PROC( )
Creates an OID that specifies a handler routine:

```
SYSCTL_ADD_PROC (struct sysctl_ctx_list * clist, struct sysctl_oid_list *
                 parent, int number, const char * name, int kind, void *
                 arg1, int arg2, int (*handler)(SYSCTL_HANDLER_ARGS),
                 const char * fmt, const char * descr);
```

The *arg1* and *arg2* parameters are passed to the *handler* routine and may be set to zero.

EXAMPLES

Creation of a node OID:

```
static struct sysctl_oid * myNode = NULL;

myNode = sysctl_add_oid (0, SYSCTL_NODE_CHILDREN(usr_ext), OID_AUTO,
                         "myNode", (int)(CTLTYPE_NODE | CTLFLAG_RD),
                         NULL, 0, NULL, "N", "This is my own node OID");
```

Alternatively with **SYSCTL_ADD_NODE( )**:

```
static struct sysctl_oid * myNode = NULL;

myNode = SYSCTL_ADD_NODE (0, SYSCTL_NODE_CHILDREN(usr_ext), OID_AUTO,
                          "myNode", (int)CTLFLAG_RD, NULL,
                          "This is my own node OID");
```

Creation of a string OID under this node:

```
    #define MAX_STRING_LENGTH       40

    static struct sysctl_oid * myString = NULL;
    static char aString[MAX_STRING_LENGTH];

    strcpy (aString, "Initial string");
    myString = sysctl_add_oid (NULL, SYSCTL_CHILDREN (myNode), OID_AUTO,
                               "myString", (int)(CTLTYPE_STRING |
CTLFLAG_RW),
                               aString, MAX_STRING_LENGTH,
                               sysctl_handle_string, "A",
                                "This is my own string OID");
```

Note the usage of the *aString* variable. Its address will be stored in the OID. Since this OID has the **CTLFLAG_RW** permission this address must be writable.

Alternatively with **SYSCTL_ADD_STRING( )**:

```
    #define MAX_STRING_LENGTH       40

    static struct sysctl_oid * myString = NULL;
    static char aString[MAX_STRING_LENGTH];

    strcpy (aString, "Initial string");
    myString = SYSCTL_ADD_STRING (NULL, SYSCTL_CHILDREN (myNode), OID_AUTO, \
                                  "myString", (int)CTLFLAG_RW, aString,     \
                                  MAX_STRING_LENGTH,                        \
                                  "This is my own string OID");
```

This node and this string will be shown as follows by the **Sysctl** command:

```
-> Sysctl "usr_ext"
usr_ext.myNode.myString: Initial string
```

RETURNS Pointer to a **sysctl_oid** structure on success, or **NULL** if an error occurred

ERRNO

EEXIST
An OID with the same name vector already exists.

EINVAL
The *parent* parameter is **NULL**.

ENOMEM
Not enough memoty available to create the OID.

SEE ALSO **kern_sysctl**, **sysctl_remove_oid( )**, **sysctl( )**

sysctl_remove_oid( )

NAME **sysctl_remove_oid( )** – remove dynamically created sysctl trees

SYNOPSIS

```
int sysctl_remove_oid
    (
    struct sysctl_oid * oidp,
    int                 del,
    int                 recurse
    )
```

DESCRIPTION This routine can be used to remove an object from the sysctl tree. It will only remove objects that were registered dynamically.

oidp
Pointer to the object that needs to be removed

del
If 0, it just de-registers this object; otherwise it frees up all resources associated with the object, such as memory for the name, and so on.

recurse
If the value is 0, this routine will return **ENOTEMPTY** for objects that are nodes and have children.

RETURNS

0
If the entry was successfully removed.

EINVAL
If *oidp* is an invalid pointer or if this object was not registered dynamically.

ENOTEMPTY
If *recurse* is 0 and the object has children.

ERRNO

N/A

SEE ALSO

kern_sysctl

sysctlbyname( )

NAME

sysctlbyname( ) – get or set the values of objects in the sysctl tree by name

SYNOPSIS

```
int sysctlbyname
    (
    char *   pName,    /* Name of object i.e "net.inet.tcp.delacktime" */
    void *   pOld,     /* Pointer to buffer to get the current values */
    size_t * pOldLen,  /* Buffer to get the size of current values */
    void *   pNew,     /* Buffer containing the new values to be set */
    size_t   newLen    /* Size of buffer containing new values */
    )
```

DESCRIPTION

This function accepts an ASCII representation of the name and internally looks up the integer name vector. Apart from that, it behaves the same as the standard **sysctl( )** function.

The information is copied into the buffer specified by *pOld*. The size of the buffer is given by the location specified by *pOldLen* before the call, and that location gives the amount of data copied after a successful call and after a call that returns with the error code **ENOMEM**. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code **ENOMEM**. If the old value is not desired, *pOld* and *pOldLen* should be set to **NULL**.

To set a new value, *pNew* is set to point to a buffer of length *newLen* from which the requested value is to be taken. If a new value is not to be set, *pNew* should be set to **NULL** and *newLen* set to 0.

For more information, please refer **sysctl( )**

RETURNS

0 on success, or -1 if an error occurred

ERRNO

Beside the errnos possibly set by the OID's non-default handler the following errnos may be set:

EPERM
An attempt is made to set a read-only value.

EINVAL
The name vector has less than two or more than **CTL_MAXNAME** elements, or the OID is not a node and has no handler, or the *newLen* size of the *pNew* buffer is too small.

ENOMEM
the *pOldLen* size of the *pOld* buffer is too small for the requested information to be stored in this buffer.

ENOENT
The OID does not exist.

EISDIR
The OID is a node without a handler so no information can be set or retreived.

ENOTDIR
One of the OID numbers in the name vector, except for the last element, does not correspond to a node OID.

SEE ALSO

kern_sysctl

sysctlnametomib( )

NAME

sysctlnametomib( ) – return the numeric representation of sysctl object

SYNOPSIS

```
int sysctlnametomib
    (
    const char * name,
    int *        mibp,
    size_t *     sizep
    )
```

DESCRIPTION

This function accepts an ASCII representation of an object in *name*, looks up the integer name vector, and returns the numeric representation in the mib array pointed to by *mibp*.

The number of elements in the mib array is given by the location specified by *sizep* before the call, and that location gives the number of entries copied after a successful call. The resulting mib and size may be used in subsequent **sysctl( )** calls to get the data associated with the requested ASCII name. This interface is intended for use by applications that want to repeatedly request the same variable (the **sysctl( )** function runs in about a third the time as the same request made via the **sysctlbyname( )** function).

RETURNS 0 on success, or -1 if an error occurred.

ERRNO

ENOMEM
the *sizep* size of the *mibp* buffer is too small for the requested information to be stored in this buffer.

ENOENT
No OID could be found for this ASCII representation of the name vector.

SEE ALSO **kern_sysctl**

tanf( )

NAME **tanf( )** – compute a tangent (ANSI)

SYNOPSIS

```
float tanf
    (
    float x  /* angle in radians */
    )
```

DESCRIPTION This routine returns the tangent of *x* in single precision. The angle *x* is expressed in radians.

RETURNS The single-precision tangent of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

tanhf( )

NAME **tanhf( )** – compute a hyperbolic tangent (ANSI)

SYNOPSIS

```
float tanhf
```

```
    (
    float x  /* number whose hyperbolic tangent is required */
    )
```

DESCRIPTION This routine returns the hyperbolic tangent of *x* in single precision.

RETURNS The single-precision hyperbolic tangent of *x*.

ERRNO Not Available

SEE ALSO **mathALib**

tarArchive( )

NAME **tarArchive( )** – archive named file/dir onto tape in tar format

SYNOPSIS

```
STATUS tarArchive
    (
    char * pTape,    /* tape device name */
    int    bfactor,  /* requested blocking factor */
    BOOL   verbose,  /* if TRUE print progress info */
    char * pName     /* file/dir name to archive */
    )
```

DESCRIPTION This function creates a UNIX compatible tar formatted archives which contain entire file hierarchies from disk file systems. Files and directories are archived with mode and time information as returned by **stat( )**.

The *tape* argument can be any tape drive device name or a name of any file that will be created if necessary, and will contain the archive. If *tape* is set to "-", standard output will be used. If *tape* is **NULL** (unspecified from Shell), the default archive file name stored in global variable *TAPE* will be used.

Each **write( )** of the archive file will be exactly *bfactor**512 bytes long, hence on tapes in variable mode, this will be the physical block size on the tape. With Fixed Mode tapes this is only a performance matter. If *bfactor* is 0, or unspecified from Shell, it will be set to the default value of 20.

The *verbose* argument is a boolean, if set to 1, will cause informative messages to be printed to standard error whenever an action is taken, otherwise, only errors are reported.

The *name* argument is the path of the hierarchy to be archived. if **NULL** (or unspecified from the Shell), the current directory path "." will be used. This is the path as seen from the target, not from the Tornado host.

All informative and error message are printed to standard error.

NOTE

Refrain from specifying absolute path names in *path*, such archives tend to be either difficult to extract or can cause unexpected damage to existing files if such exist under the same absolute name.

There is no way of specifying a number of hierarchies to dump.

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

tarLib

tarExtract( )

NAME

tarExtract( ) – extract all files from a tar formatted tape

SYNOPSIS

```
STATUS tarExtract
    (
    char * pTape,     /* tape device name */
    int    bfactor,  /* requested blocking factor */
    BOOL   verbose   /* if TRUE print progress info */
    )
```

DESCRIPTION

This is a UNIX-tar compatible utility that extracts entire file hierarchies from tar-formatted archive. The files are extracted with their original names and modes. In some cases a file cannot be created on disk, for example if the name is too long for regular DOS file name conventions, in such cases entire files are skipped, and this program will continue with the next file. Directories are created in order to be able to create all files on tape.

The *tape* argument may be any tape device name or file name that contains a tar formatted archive. If *tape* is equal "-", standard input is used. If *tape* is **NULL** (or unspecified from Shell) the default archive file name stored in global variable *TAPE* is used.

The *bfactor* dictates the blocking factor the tape was written with. If 0, or unspecified from the shell, a default of 20 is used.

The *verbose* argument is a boolean, if set to 1, will cause informative messages to be printed to standard error whenever an action is taken, otherwise, only errors are reported.

All informative and error message are printed to standard error.

There is no way to selectively extract tar archives with this utility. It extracts entire archives.

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO **tarLib**

tarToc( )

NAME **tarToc( )** – display all contents of a tar formatted tape

SYNOPSIS

```
STATUS tarToc
    (
    char * tape,    /* tape device name */
    int    bfactor  /* requested blocking factor */
    )
```

DESCRIPTION This is a UNIX-tar compatible utility that displays entire file hierarchies from tar-formatted media, e.g. tape.

The *tape* argument may be any tape device name or file name that contains a tar formatted archive. If *tape* is equal "-", standard input is used. If *tape* is **NULL** (or unspecified from Shell) the default archive file name stored in global variable *TAPE* is used.

The *bfactor* dictates the blocking factor the tape was written with. If 0, or unspecified from Shell, default of 20 is used.

Archive contents are displayed on standard output, while all informative and eror message are printed to standard error.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **tarLib**

taskActivate( )

NAME **taskActivate( )** – activate a task that has been initialized

SYNOPSIS

```
STATUS taskActivate
    (
    int tid  /* task ID of task to activate */
    )
```

DESCRIPTION This routine activates tasks created by **taskInit( )**. Without activation, a task is ineligible for CPU allocation by the scheduler.

The *tid* (task ID) argument is simply the address of the **WIND_TCB** for the task (the **taskInit( )** *pTcb* argument), cast to an integer:

```
tid = (int) pTcb;
```

The **taskSpawn( )** routine is built from **taskActivate( )** and **taskInit( )**. Tasks created by **taskSpawn( )** do not require explicit task activation.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task cannot be activated.

ERRNO **S_objLib_OBJ_ID_ERROR**
The *tid* parameter is an invalid task ID.

SEE ALSO **taskLib**, **taskInit( )**

taskClose( )

NAME **taskClose( )** – close a task

SYNOPSIS

```
STATUS taskClose
    (
    int tid  /* task to close */
    )
```

DESCRIPTION This routine closes a task. It decrements the task's reference counter.

This routine does not delete a task. **taskDelete( )** should be called to terminate and delete a task.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if tid is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**
Task ID is invalid.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **taskOpen**, **taskOpen( )**

taskCpuAffinityGet( )

NAME **taskCpuAffinityGet( )** – get the CPU affinity of a task

SYNOPSIS
```
STATUS taskCpuAffinityGet
    (
    int       tid,       /* task ID */
    cpuset_t* pAffinity  /* address to store task's affinity */
    )
```

DESCRIPTION This routine provides the caller with the CPU affinity of task *tid*. This affinity is represented using a CPU set that is copied in the user supplied *pAffinity*. Passing a null task ID causes the affinity of the caller to be provided. If tid has no affinity the resulting pAffinity CPU set contains no CPU index. If tid has an affinity, the resulting pAffinity CPU set is identical to the CPU set that was passed on the last invocation of **taskCpuAffinitySet( )** for that task. This behaviour also applies when calling this routine in the uniprocessor version of VxWorks.

This routine must not be called from interrupt level.

The following code example shows how a task can determine if it has an affinity:

```
{
cpuset_t affinity;

if (taskCpuAffinityGet (0, &affinity) == OK)
    {
    if (CPUSET_ISZERO(affinity))
        {
        /* Task has no affinity */
        }
    else
        {
        /* Task has an affinity */
        }
    }
}
```

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK** or **ERROR** if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**

S_intLib_NOT_ISR_CALLABLE

SEE ALSO **taskLib**, **taskCpuAffinitySet( )**, cpuset

taskCpuAffinitySet( )

NAME **taskCpuAffinitySet( )** – set the CPU affinity of a task

SYNOPSIS

```
STATUS taskCpuAffinitySet
    (
    int      tid,         /* task ID */
    cpuset_t newAffinity  /* new affinity set */
    )
```

DESCRIPTION This routine sets the CPU affinity of task *tid* to the CPU specified in *newAffinity*. From that point on the scheduler ensures the task is only executed on the specified CPU. Passing a tid equal to zero causes an affinity to be set for the calling task. Should the tid argument refer to a task presently running on a CPU other than the one listed in the newAffinity argument, this routine causes the task to cease execution and be rescheduled, based on its priority, on the CPU it has an affinity for. Therefore calling this routine can cause a scheduling event to take place. Calling this routine with an empty CPU set as the newAffinity argument effectively removes any affinity for task tid. If the CPU set identifies a CPU index that is not one of the CPUs configured in the system or if the set contains more than one CPU an error is returned. Once a task has an affinity set, all other tasks it creates have the same affinity except for the case where the child task is the init task of an RTP created using the **rtpSpawn( )** API.

Calling this routine in the uniprocessor version of VxWorks is permitted but the newAffinity argument must specify that CPU 0 is the one the task has an affinity for. This action has no effect whatsoever on the scheduling of the task. The only visible effect on uniprocessor VxWorks is that a subsequent call to **taskCpuAffinityGet( )** would indicate the task has an affinity to CPU 0.

This routine must not be called from interrupt level.

The following sample code illustrates the sequence to set the affinity of a newly created task to CPU 1:

```
STATUS affinitySetExample (void)
   {
   cpuset_t affinity;
   int tid;
   /* Create the task but only activate it after setting its affinity */
   tid = taskCreate ("myCpu1Task", 100, 0, 5000, printf,
                     (int) "myCpu1Task executed on CPU 1 !", 0, 0, 0,
                     0, 0, 0, 0, 0, 0);

   if (tid == NULL)
       return (ERROR);

   /* Clear the affinity CPU set and set index for CPU 1 */
   CPUSET_ZERO (affinity);
   CPUSET_SET  (affinity, 1);

   if (taskCpuAffinitySet (tid, affinity) == ERROR)
       {
       /* Ooops, looks like we're running on a uniprocessor */
       taskDelete (tid);
       return (ERROR);
       }

   /* Now let the task run on CPU 1 */
   taskActivate (tid);

   return (OK);
   }
```

The following example shows how a task can remove its affinity to a CPU:

```
{
cpuset_t affinity;

CPUSET_ZERO (affinity);

taskCpuAffinitySet (0, affinity);
}
```

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if the task ID or affinity is invalid.

ERRNO

S_taskLib_ILLEGAL_OPERATION

S_objLib_OBJ_ID_ERROR

S_intLib_NOT_ISR_CALLABLE

SEE ALSO

taskLib, **taskCpuAffinityGet( )**, **vxCpuConfiguredGet( )**, **cpuset**

taskCpuLock( )

NAME
taskCpuLock( ) – disable local CPU task rescheduling

SYNOPSIS
```
STATUS taskCpuLock (void)
```

DESCRIPTION
This routine disables scheduling on the CPU the calling task is running on. This effectively prevents any other task from running on the local CPU and prevents the calling task from migrating to another CPU until the lock is released by calling **taskCpuUnlock( )**. This could prove useful when used in conjunction with the **vxCpuIndexGet( )** to ensure a CPU index stays valid for a short period of time while a per-CPU object needs to be read or modified. This routine can be recursively called but there is no effect on scheduling other than an equal number of calls to **taskCpuUnlock( )** needs to be done to re-enable scheduling on the local CPU. Execution on other CPUs in the SMP system is not affected by this routine. Because of this behaviour this routine is not a suitable task mutual exclusion mechanism unless all tasks participating in the mutual exclusion scenario have a single CPU affinity for the very same CPU. A task that calls a blocking API such as **semTake( )** after calling **taskCpuLock( )** constitutes an error condition that results in an error to be returned and reported through ED&R regardless of the fact the task may not block at all. When a task is in a task locked state, its priority cannot be modified by a task or ISR running on another CPU nor can it be suspended or stopped by a task or ISR running on another CPU. Calling this routine on the uniprocessor version of VxWorks is equivalent to calling **taskLock( )**.

This routine is not callable from interrupt level. This is not enforced and it is the user's responsibility to adhere to this restriction.

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS
OK, or **ERROR**

ERRNO
S_taskLib_ILLEGAL_OPERATION

S_objLib_OBJ_ID_ERROR

SEE ALSO
taskLib, **taskCpuUnlock( )**, **vxCpuIndexGet( )**

taskCpuUnlock( )

NAME

taskCpuUnlock( ) – enable local CPU task rescheduling

SYNOPSIS

```
STATUS taskCpuUnlock (void)
```

DESCRIPTION

This routine removes the lock established using **taskCpuLock( )**. It re-enables context task switching on the CPU the calling task is running on. Calling this routine on the uniprocessor version of VxWorks is equivalent to calling **taskUnlock( )**.

This routine is not callable from interrupt level. This is not enforced and it is the user's responsibility to adhere to this restriction.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR**

ERRNO

S_taskLib_ILLEGAL_OPERATION

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, **taskCpuLock( )**

taskCreate( )

NAME

taskCreate( ) – allocate and initialize a task without activation

SYNOPSIS

```
int taskCreate
    (
    char *  name,       /* name of new task (stored at pStackBase) */
    int     priority,   /* priority of new task */
    int     options,    /* task option word */
    int     stackSize,  /* size (bytes) of stack needed */
    FUNCPTR entryPt,    /* entry point of new task */
    int     arg1,       /* 1st of 10 req'd args to pass to entryPt */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9,
    int     arg10
    )
```

DESCRIPTION This routine creates, but does not activate, a new task with a specified priority and options and returns a system-assigned ID. Activate the newly created task by invoking **taskActivate( )**.

To create **and** activate a new task, use the **taskSpawn( )** routine instead of **taskCreate( )**.

See **taskSpawn( )** for an explanation of all the parameters.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS Task ID, or **NULL** if out of memory or unable to create task.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to spawn the task.

S_taskLib_ILLEGAL_PRIORITY
A priority outside the range 0 to 255 was specified.

S_taskLib_ILLEGAL_OPTIONS
The following illegal options were set: **VX_DEALLOC_STACK**, **VX_DEALLOC_EXC_STACK**, or **VX_DEALLOC_TCB** for **taskCreate( )**.

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified

SEE ALSO **taskLib**, **taskSpawn( )**, **taskCreat( )**, **taskActivate( )**, The VxWorks Programmer's Guide

taskCreateHookAdd( )

NAME **taskCreateHookAdd( )** – add a routine to be called at every task create

SYNOPSIS

```
STATUS taskCreateHookAdd
    (
    FUNCPTR createHook  /* routine to be called when a task is created */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called whenever a task is created. Upon creation, all routines specified by **taskCreateHookAdd( )** will be called in the context of the creating task, so any objects created by a task create hook will be owned by the caller's RTP rather than the newly created task's RTP. To set the ownership of newly created objects to the new task's RTP, **objOwnerSet( )** should be used, e.g.

objOwnerSet (createdObjId, pNewTcb->rtpId)

The routine should be declared as follows:

```
void createHook
    (
    WIND_TCB *pNewTcb   /* pointer to new task's TCB */
    )
```

RETURNS **OK**, or **ERROR** if the table of task create routines is full.

ERRNO Not Available

SEE ALSO **taskHookLib**, **taskCreateHookDelete( )**

taskCreateHookDelete( )

NAME **taskCreateHookDelete( )** – delete a previously added task create routine

SYNOPSIS

```
STATUS taskCreateHookDelete
    (
    FUNCPTR createHook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each task create.

RETURNS **OK**, or **ERROR** if the routine is not in the table of task create routines.

ERRNO Not Available

SEE ALSO **taskHookLib**, **taskCreateHookAdd( )**

taskCreateHookShow( )

NAME **taskCreateHookShow( )** – show the list of task create routines

SYNOPSIS

```
void taskCreateHookShow (void)
```

DESCRIPTION This routine shows all the task create routines installed in the task create hook table, in the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **taskHookShow**, **taskCreateHookAdd( )**

taskDelay( )

NAME **taskDelay( )** – delay a task from executing

SYNOPSIS

```
STATUS taskDelay
    (
    int ticks  /* number of ticks to delay task */
    )
```

DESCRIPTION This routine causes the calling task to relinquish the CPU for the duration specified (in ticks). This is commonly referred to as manual rescheduling, but it is also useful when waiting for some external condition that does not have an interrupt associated with it.

If the calling task receives a signal that is not being blocked or ignored, **taskDelay( )** returns **ERROR** and sets **errno** to **EINTR** after the signal handler is run.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if called from interrupt level or if the calling task receives a signal that is not blocked or ignored.

ERRNO **S_intLib_NOT_ISR_CALLABLE**

EINTR

SEE ALSO **taskLib**

taskDelete( )

NAME

taskDelete( ) – delete a task

SYNOPSIS

```
STATUS taskDelete
    (
    int tid  /* task ID of task to delete */
    )
```

DESCRIPTION

This routine causes a specified task to cease to exist and deallocates the stack and **WIND_TCB** memory resources. Upon deletion, all routines specified by **taskDeleteHookAdd( )** will be called in the context of the deleting task. This routine is the companion routine to **taskSpawn( )**.

WARNING

Deleting individual user tasks, as opposed to deleting the entire RTP, may result in unpredictable RTP behavior. The deletion of individual user tasks should only be performed for debugging purposes.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if the task cannot be deleted.

ERRNO

S_intLib_NOT_ISR_CALLABLE

S_objLib_OBJ_DELETED

S_objLib_OBJ_UNAVAILABLE

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, **excLib**, **taskDeleteHookAdd( )**, **taskSpawn( )**, the VxWorks programmer's guides

taskDeleteForce( )

NAME

taskDeleteForce( ) – delete a task without restriction

SYNOPSIS

```
STATUS taskDeleteForce
    (
    int tid  /* task ID of task to delete */
    )
```

DESCRIPTION This routine deletes a task even if the task is protected from deletion. It is similar to **taskDelete( )**. Upon deletion, all routines specified by **taskDeleteHookAdd( )** will be called in the context of the deleting task.

CAVEATS This routine is intended as a debugging aid, and is generally inappropriate for applications. Disregarding a task's deletion protection could leave the the system in an unstable state or lead to system deadlock.

The system does not protect against simultaneous **taskDeleteForce( )** calls. Such a situation could leave the system in an unstable state.

Deleting individual user tasks, as opposed to deleting the entire RTP, may result in unpredictable RTP behavior.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task cannot be deleted.

ERRNO **S_intLib_NOT_ISR_CALLABLE**

S_objLib_OBJ_DELETED

S_objLib_OBJ_UNAVAILABLE

S_objLib_OBJ_ID_ERROR

SEE ALSO **taskLib**, **taskDeleteHookAdd( )**, **taskDelete( )**

taskDeleteHookAdd( )

NAME **taskDeleteHookAdd( )** – add a routine to be called at every task delete

SYNOPSIS

```
STATUS taskDeleteHookAdd
    (
    FUNCPTR deleteHook  /* routine to be called when a task is deleted */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called whenever a task is deleted. Upon deletion, all routines specified by **taskDeleteHookAdd( )** will be called in the context of the deleting task.

The routine should be declared as follows:

```
    void deleteHook
```

```
        (
        WIND_TCB *pTcb      /* pointer to deleted task's WIND_TCB */
        )
```

RETURNS **OK**, or **ERROR** if the table of task delete routines is full.

ERRNO Not Available

SEE ALSO **taskHookLib**, **taskDeleteHookDelete( )**

taskDeleteHookDelete( )

NAME **taskDeleteHookDelete( )** – delete a previously added task delete routine

SYNOPSIS

```
STATUS taskDeleteHookDelete
    (
    FUNCPTR deleteHook  /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each task delete.

RETURNS **OK**, or **ERROR** if the routine is not in the table of task delete routines.

ERRNO Not Available

SEE ALSO **taskHookLib**, **taskDeleteHookAdd( )**

taskDeleteHookShow( )

NAME **taskDeleteHookShow( )** – show the list of task delete routines

SYNOPSIS

```
void taskDeleteHookShow (void)
```

DESCRIPTION This routine shows all the delete routines installed in the task delete hook table, in the order in which they were installed. Note that the delete routines will be run in reverse of the order in which they were installed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **taskHookShow**, **taskDeleteHookAdd( )**

taskExit( )

NAME **taskExit( )** – exit a task

SYNOPSIS

```
void taskExit
    (
    int exitCode        /* code stored in TCB for delete hooks */
    )
```

DESCRIPTION This routine is called by a task to cease to exist as a task. The *exitCode* parameter will be stored in the **WIND_TCB** for possible use by the delete hooks, or post-mortem debugging.

This function is currently aliased to **exit( )**, and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS N/A

ERRNO N/A

SEE ALSO **taskLib**, **taskDelete( )**, *"American National Standard for Information Systems - "Programming Language - C, ANSI X3.159-1989: Input/Output (***stdlib.h**), ", the VxWorks programmer's guides

taskHookShowInit( )

NAME **taskHookShowInit( )** – initialize the task hook show facility

SYNOPSIS

```
void taskHookShowInit (void)
```

DESCRIPTION This routine 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.

RETURNS N/A

ERRNO Not Available

SEE ALSO **taskHookShow**

taskIdDefault( )

NAME **taskIdDefault( )** – set the default task ID

SYNOPSIS

```
int taskIdDefault
    (
    int tid  /* user supplied task ID; if 0, return default */
    )
```

DESCRIPTION This routine maintains a global default task ID. This ID is used by libraries that want to allow a task ID argument to take on a default value if the user did not explicitly supply one.

If *tid* is not zero (i.e., the user did specify a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (i.e., the user did not specify a task ID), the default ID is not changed and its value is returned. Thus the value returned is always the last task ID the user specified.

RETURNS The most recent non-zero task ID.

ERRNOS N/A

SEE ALSO **taskInfo**, **dbgLib**, **windsh**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

taskIdListGet( )

NAME **taskIdListGet( )** – get a list of active task IDs

SYNOPSIS

```
int taskIdListGet
```

```
    (
    int idList[],  /* array of task IDs to be filled in */
    int maxTasks   /* max tasks <idList> can accommodate */
    )
```

DESCRIPTION This routine provides the calling task with a list of all active tasks. An unsorted list of task IDs for no more than *maxTasks* tasks is put into *idList*.

The provided list is a snapshot of the system. There is no guarantee that this snapshot will still represent the state of the system by the time execution returns to the caller. This is especially true on VxWorks SMP because of the concurrent execution environment.

This routine is provided if the **INCLUDE_TASK_LIST** component is present in the configuration.

RETURNS The number of tasks put into the ID list.

ERRNO N/A

SEE ALSO **taskInfo**, taskNameToId

taskIdSelf( )

NAME **taskIdSelf( )** – get the task ID of a running task

SYNOPSIS

```
int taskIdSelf (void)
```

DESCRIPTION This routine gets the task ID of the calling task. The task ID will be invalid if called at interrupt level.

RETURNS The task ID of the calling task.

ERRNO N/A

SEE ALSO **taskLib**

taskIdVerify( )

NAME

taskIdVerify( ) – verify the existence of a task

SYNOPSIS

```
STATUS taskIdVerify
    (
    int tid  /* task ID */
    )
```

DESCRIPTION

This routine verifies the existence of a specified task by validating the specified ID as a task ID. Note that an exception occurs if the task ID parameter points to an address not located in physical memory.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib

taskInfoGet( )

NAME

taskInfoGet( ) – get information about a task

SYNOPSIS

```
STATUS taskInfoGet
    (
    int         tid,       /* ID of task for which to get info */
    TASK_DESC * pTaskDesc  /* task descriptor to be filled in */
    )
```

DESCRIPTION

This routine fills in a specified task descriptor (**TASK_DESC**) for a specified task. The information in the task descriptor is, for the most part, a copy of information kept in the task control block (**WIND_TCB**). The **TASK_DESC** structure is useful for common information and avoids dealing directly with the unwieldy **WIND_TCB**.

NOTE

Examination of WIND_TCBs should be restricted to debugging aids.

RETURNS

OK, or **ERROR** if the task ID is invalid or access to task info denied.

ERRNO

N/A

SEE ALSO

taskShow

taskInit( )

NAME

taskInit( ) – initialize a task with a stack at a specified address

SYNOPSIS

```
STATUS taskInit
    (
    FAST WIND_TCB *pTcb,        /* address of new task's TCB */
    char *        name,         /* name of new task (stored at pStackBase) */
    int           priority,     /* priority of new task */
    int           options,      /* task option word */
    char *        pStackBase,   /* base of new task's execution stack */
    int           stackSize,    /* size (bytes) of stack needed */
    FUNCPTR       entryPt,      /* entry point of new task */
    int           arg1,         /* 1st of 10 req'd args to pass to entryPt */
    int           arg2,
    int           arg3,
    int           arg4,
    int           arg5,
    int           arg6,
    int           arg7,
    int           arg8,
    int           arg9,
    int           arg10
    )
```

DESCRIPTION

This routine initializes user-specified regions of memory for a task stack and control block instead of allocating them from memory as **taskSpawn( )** does. This routine will utilize the specified pointers to the **WIND_TCB** and stack as the components of the task. This allows, for example, the initialization of a static **WIND_TCB** variable. It also allows for special stack positioning as a debugging aid.

As in **taskSpawn( )**, a task may be given a name. While **taskSpawn( )** automatically names unnamed tasks, **taskInit( )** permits the existence of tasks without names. The task ID required by other task routines is simply the address *pTcb*, cast to an integer.

Unlike **taskSpawn( )**, **taskInit( )** allows one to control the activation of the **VX_DEALLOC_STACK** bit in *options*; **taskSpawn( )** always sets this bit. Setting this bit causes the stack to be automatically deallocated when a **taskDelete( )** is performed, or when a return command is issued from the entry function.

It is not recommended to use the **VX_DEALLOC_STACK** option for **taskInit( )** if RTP or **KERNEL_HARDENING** is configured into the system. A system configured with RTP or **KERNEL_HARDENING** expects the task stacks to have guard zones, governed by the configuration parameters **TASK_KERNEL_EXEC_STACK_UNDERFLOW_SIZE** and **TASK_KERNEL_EXEC_STACK_OVERFLOW_SIZE**. Setting the **VX_DEALLOC_STACK** option but not set the appropriate guard zones in these configuration might cause corruption in the system. If corruption occurs, use the Error Detection and Reporting mechanism to help you detect the corruption.

If it is necessary to use the **VX_DEALLOC_STACK** option in the above configurations, the user is responsible for setting up the guard zones corresponding to the same value of the configuration parameters or the user can specify the **VX_NO_PROTECT** option for the task.

The *pStackBase* parameter specifies the base of the execution stack. The stack may grow up or down from *pStackBase* depending on the target architecture. The caller is responsible for setting up any guard zones around the specified stack area. The following code fragment illustrates how to specify the stack base location:

For architectures where the stack grows down:

```
pStackMem = (char *) memalign (_STACK_ALIGN_SIZE, stackSize);

if (pStackMem != NULL)
    status = taskInit ( ... , pStackMem + stackSize, stackSize, ... );
```

For architectures where the stack grows up:

```
pStackMem = (char *) memalign (_STACK_ALIGN_SIZE, stackSize);

if (pStackMem != NULL)
    status = taskInit ( ... , pStackMem, stackSize, ... );
```

Please note that **memalign( )** is used in the above code fragment for illustrative purposes only since it's a well-known API. The stack memory can be obtained by any mechanism that ensures allocation of aligned memory region.

The *stackSize* parameter specifies the size in bytes of the execution stack area. This API does not check against illegal stack size, since it is assumed that the user has allocated the stack memory with a valid stack size, before calling this API.

It is assumed that the caller passes valid pointers for *pTcb*, *pStackBase* and *entryPt* while calling this API. No validity checks for these parameters are done here.

Other arguments are the same as in **taskSpawn( )**. Unlike **taskSpawn( )**, **taskInit( )** does not activate the task. This must be done by calling **taskActivate( )** after calling **taskInit( )**.

Normally, tasks should be started using **taskSpawn( )** rather than **taskInit( )**, except when additional control is required for task memory allocation or a separate task activation is desired.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

NOTE For future releases of VxWorks, each task has an exception stack. To continue providing the **taskInit( )** API, this routine now carves memory for the exception stack of a task . To use a specific region of memory for the exception stack, use the routine **taskInitExcStk( )** instead.

RETURNS **OK**, or **ERROR** if the task cannot be initialized.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine is not callable from an ISR.

S_taskLib_ILLEGAL_PRIORITY
Priority specified is not within 0-255.

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, **taskActivate( )**, **taskSpawn( )**, **taskInitExcStk( )**

taskInitExcStk( )

NAME

taskInitExcStk( ) – initialize a task with stacks at specified addresses

SYNOPSIS

```
STATUS taskInitExcStk
    (
    FAST WIND_TCB *pTcb,           /* address of new task's TCB */
    char *        name,            /* name of new task (stored at pStackBase)
*/
    int           priority,        /* priority of new task */
    int           options,         /* task option word */
    char *        pStackBase,      /* base of new task's execution stack */
    int           stackSize,       /* size (bytes) of stack needed */
    char *        pExcStackBase,   /* base of new task's exception stack */
    int           excStackSize,    /* size (bytes) of exception stack needed
*/
    FUNCPTR       entryPt,         /* entry point of new task */
    int           arg1,            /* first of ten task args to pass to func
*/
    int           arg2,
    int           arg3,
    int           arg4,
    int           arg5,
    int           arg6,
    int           arg7,
    int           arg8,
    int           arg9,
    int           arg10
    )
```

DESCRIPTION

This routine initializes user-specified regions of memory for a task stack, exception stack, and control block instead of allocating them from memory as **taskSpawn( )** does. This routine will utilize the specified pointers to the **WIND_TCB** and stacks as the components of the task. This allows, for example, the initialization of a static **WIND_TCB** variable. It also allows for special stack positioning as a debugging aid.

As in **taskSpawn( )**, a task may be given a name. While **taskSpawn( )** automatically names unnamed tasks, **taskInitExcStk( )** permits the existence of tasks without names. The task ID required by other task routines is simply the address *pTcb*, cast to an integer.

Unlike **taskSpawn( )**, **taskInitExcStk( )** allows one to control the activation of both the **VX_DEALLOC_STACK** and **VX_DEALLOC_EXC_STACK** bits in *options*; Setting the **VX_DEALLOC_STACK** bit causes the stack (aka execution stack) to be automatically deallocated when a **taskDelete( )** is performed, or when a return command is issued from the entry function. Setting the **VX_DEALLOC_EXC_STACK** bit causes the exception stack to be automatically deallocated when a **taskDelete( )** is performed, or when a return command is issued from the the entry function.

It is not recommended to use the **VX_DEALLOC_STACK** option for **taskInit( )** if RTP or **KERNEL_HARDENING** is configured into the system. A system configured with RTP or **KERNEL_HARDENING** expects the task stacks to have guard zones, governed by the configuration parameters **TASK_KERNEL_EXEC_STACK_UNDERFLOW_SIZE** and **TASK_KERNEL_EXEC_STACK_OVERFLOW_SIZE**. Setting the **VX_DEALLOC_STACK** option but not set the appropriate guard zones in these configuration might cause corruption in the system. If corruption occurs, use the Error Detection and Reporting mechanism to help you detect the corruption.

If it is necessary to use the **VX_DEALLOC_STACK** option in the above configurations, the user is responsible for setting up the guard zones corresponding to the same value of the configuration parameters or the user can specify the **VX_NO_PROTECT** option for the task.

The *pStackBase* parameter specifies the base of the execution stack, and *pExcStackBase* specified the base of the exception stack. The stacks may grow up or down from *pStackBase*/*pExcStackBase* depending on the target architecture. The caller is responsible for setting up any guard zones around the specified stack areas. The following code fragment illustrates how to specify the stack base location:

For architectures where the stack grows down:

```
pStackMem    = (char *) memalign (_STACK_ALIGN_SIZE, stackSize);
pExcStackMem = (char *) memalign (_STACK_ALIGN_SIZE, excStackSize);

if ((pStackMem != NULL) && (pExcStackMem != NULL))
    status = taskInitExcStk ( ... , pStackMem + stackSize,
                                     stackSize,
                                     pExcStackMem + excStackSize,
                                     excStackSize, ... );
```

For architectures where the stack grows up:

```
pStackMem    = (char *) memalign (_STACK_ALIGN_SIZE, stackSize);
pExcStackMem = (char *) memalign (_STACK_ALIGN_SIZE, excStackSize);

if ((pStackMem != NULL) && (pExcStackMem != NULL))
    status = taskInitExcStk ( ... , pStackMem,
                                    stackSize,
                                    pExcStackMem,
                                    excStackSize, ... );
```

Please note that **memalign( )** is used in the above code fragment for illustrative purposes only since it's a well-known API. Typically, the stack memory would be obtained by some other mechanism.

Other arguments are the same as in **taskSpawn( )**. Unlike **taskSpawn( )**, **taskInitExcStk( )** does not activate the task. This must be done by calling **taskActivate( )** after calling **taskInitExcStk( )**.

Normally, tasks should be started using **taskSpawn( )** rather than **taskInit( )** or **taskInitExcStk( )**, except when additional control is required for task memory allocation or a separate task activation is desired.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if the task cannot be initialized.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine is not callable from an ISR.

S_taskLib_ILLEGAL_PRIORITY
Priority specified is not within 0-255.

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, **taskActivate( )**, **taskSpawn( )**, **taskInit( )**

taskIsPended( )

NAME

taskIsPended( ) – check if a task is pended

SYNOPSIS

```
BOOL taskIsPended
    (
    int tid  /* task ID */
    )
```

DESCRIPTION

This routine tests the status field of a task to determine if it is pended. No indication is given regarding the timeout, if any, associated with the pending operation.

RETURNS

TRUE if the task is pended, otherwise **FALSE**.

ERRNOS

N/A

SEE ALSO **taskInfo**

taskIsReady( )

NAME **taskIsReady( )** – check if a task is ready to run

SYNOPSIS

```
BOOL taskIsReady
    (
    int tid  /* task ID */
    )
```

DESCRIPTION This routine tests the status field of a task to determine if it is ready to run.

RETURNS **TRUE** if the task is ready, otherwise **FALSE**.

ERRNOS N/A

SEE ALSO **taskInfo**

taskIsStopped( )

NAME **taskIsStopped( )** – check if a task is stopped by the debugger

SYNOPSIS

```
BOOL taskIsStopped
    (
    int tid  /* task ID */
    )
```

DESCRIPTION This routine tests the status field of a task to determine if it is stopped by the debugger.

RETURNS **TRUE** if the task is stopped by the debugger, otherwise **FALSE**.

ERRNOS N/A

/NOMANUAL

SEE ALSO **taskInfo**

taskIsSuspended( )

NAME

taskIsSuspended( ) – check if a task is suspended

SYNOPSIS

```
BOOL taskIsSuspended
    (
    int tid  /* task ID */
    )
```

DESCRIPTION

This routine tests the status field of a task to determine if it is suspended.

RETURNS

TRUE if the task is suspended, otherwise **FALSE**.

ERRNOS

N/A

SEE ALSO

taskInfo

taskKill( )

NAME

taskKill( ) – send a signal to a task

SYNOPSIS

```
int taskKill
    (
    int tid,
    int signo
    )
```

DESCRIPTION

This routine sends a signal *signo* to the task specified by *tid*. This function is currently aliased to **kill( )**, and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

RETURNS

OK (0), or **ERROR** (-1) if the task ID or signal number is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib, **kill( )**

taskLock( )

NAME

taskLock( ) – disable task rescheduling

SYNOPSIS

```
STATUS taskLock
    (
    void
    )
```

DESCRIPTION

This routine disables task context switching. The task that calls this routine will be the only task that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. Typically this call is paired with **taskUnlock( )**; together they surround a critical section of code. These preemption locks are implemented with a counting variable that allows nested preemption locks. Preemption will not be unlocked until **taskUnlock( )** has been called as many times as **taskLock( )**.

This routine does not lock out interrupts; use **intLock( )** to lock out interrupts.

A **taskLock( )** is preferable to **intLock( )** as a means of mutual exclusion, because interrupt lock-outs add interrupt latency to the system.

A **semTake( )** is preferable to **taskLock( )** as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the system.

WARNINGS

The **taskLock( )** routine is not callable from interrupt service routines.

Invoking a VxWorks system routine with preemption locked may result in preemption being unlocked for an unspecified period of time. If the called routine blocks or suspends the calling task, the scheduler will always select the highest priority ready task to execute (or become idle) regardless of whether the task has locked preemption via **taskLock( )**.

The preemption lock state is an attribute of a task, i.e. it's part of the task context. Thus, if a task disables preemption and subsequently invokes a VxWorks system routine that causes the calling task to block or cause a higher priority task to be ready, the preemption lock state will be restored when the task is later rescheduled for execution.

SMP CONSIDERATIONS

This API is not available in VxWorks SMP.

RETURNS

N/A

RETURNS

OK or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR

S_intLib_NOT_ISR_CALLABLE

SEE ALSO

taskLib, **taskUnlock( )**, **taskCpuLock( )**, **intLock( )**, **taskSafe( )**, **semTake( )**

taskName( )

NAME

taskName( ) – get the name associated with a task ID

SYNOPSIS

```
char * taskName
    (
    int tid  /* ID of task whose name is to be found */
    )
```

DESCRIPTION

This routine returns a pointer to the name of a task of a specified ID, if the task has a name. If the task has no name, it returns an empty string.

RETURNS

A pointer to the task name, or **NULL** if the task ID is invalid.

ERRNOS

N/A

SEE ALSO

taskInfo

taskNameToId( )

NAME

taskNameToId( ) – look up the task ID associated with a task name

SYNOPSIS

```
int taskNameToId
    (
    char * name  /* task name to look up */
    )
```

DESCRIPTION

This routine returns the ID of the task matching a specified name. Referencing a task in this way is inefficient, since it involves a search of the task list.

This routine is provided if the **INCLUDE_TASK_LIST** component is present in the configuration.

RETURNS

The task ID, or **ERROR** if the task is not found.

ERRNO

S_taskLib_NAME_NOT_FOUND

SEE ALSO

taskInfo, taskName

taskOpen( )

NAME

taskOpen( ) – open a task

SYNOPSIS

```
int taskOpen
    (
    const char * name,        /* task name - default name will be chosen */
    int          priority,    /* task priority */
    int          options,     /* VX_ task option bits */
    int          mode,        /* object management mode bits */
    char *       pStackBase,  /* base of new task's execution stack */
    int          stackSize,   /* execution stack size */
    void       * context,     /* context value */
    FUNCPTR      entryPt,     /* application entry point */
    int          arg1,        /* 1st of 10 req'd args to pass to entryPt */
    int          arg2,
    int          arg3,
    int          arg4,
    int          arg5,
    int          arg6,
    int          arg7,
    int          arg8,
    int          arg9,
    int          arg10
    )
```

DESCRIPTION

The **taskOpen( )** API is the most general purpose task creation routine. It can also be used to obtain a task ID to an already existing task, typically a public task with an RTP. It searches the task name space for a matching task. If a matching task is found, it returns the task ID of the matched task. If a matching task is not found but the **OM_CREATE** flag is specified in the *mode* parameter, then it creates a task. This routine is not ISR callable.

There are two name spaces available in which **taskOpen( )** can perform the search. The name space searched is dependent upon the first character in the *name* parameter. When this character is a forward slash **/**, the **public** name space is searched; otherwise the **private** name space is searched. Similarly, if a task is created, the *name*'s first character specifies the name space that contains the task.

Unlike other objects in VxWorks, private task names are not unique. Thus a search on a private name space finds the first matching task. However, this task may not be the only task with the specified name. Public task names on the other hand, are unique.

A description of the **taskOpen( )** arguments follows:

name

This is a mandatory argument. Unlinke **taskSpawn( )**, **NULL** or empty strings are not allowed when using this routine. The task's name appears in various kernel shell facilities such as **i( )**. The name may be of arbitrary length and content. Public task names are unique, private task names are not.

priority

The VxWorks kernel schedules tasks on the basis of priority. Tasks may have priorities ranging from 0 (highest) to 255 (lowest). The priority of a task in VxWorks is dynamic, and the priority of an existing task can be changed using **taskPrioritySet( )**. Also, a task can inherit a priority as a result of the acquisition of a priority-inversion-safe mutex semaphore.

options

Bits in the options argument may be set to run with the following modes:

VX_UNBREAKABLE	do not allow breakpoint debugging
VX_FP_TASK	execute with floating-point coprocessor support
VX_ALTIVEC_TASK	execute with Altivec support (PowerPC only)
VX_SPE_TASK	execute with SPE support (PowerPC only)
VX_DSP_TASK	execute with DSP support (SuperH only)
VX_PRIVATE_ENV	the task has a private environment area
VX_NO_STACK_FILL	do not fill the stack with 0xee (for debugging)
VX_TASK_NOACTIVATE	do not activate the task upon creation
VX_NO_STACK_PROTECT	do not provide overflow/underflow stack protection, stack remains executable.

mode

This parameter specifies the various object management attribute bits as follows:

OM_CREATE
Create a new task if a matching task name is not found.

OM_EXCL
When set jointly with **OM_CREATE**, create a new task immediately without attempting to open an existing task. The call fails if the task is public and its name causes a name clash. This flag has no effect if the **OM_CREATE** attribute is not specified.

OM_DELETE_ON_LAST_CLOSE
This bit is ignored on tasks because it would allow a task to be deleted from another RTP.

pStackBase

Base of the execution stack. When a **NULL** pointer is specified, the kernel allocates a page-aligned stack area.

The stack may grow up or down from *pStackBase* depending on the target architecture. The caller is responsible for setting up any guard zones around the specified stack area. The following code fragment illustrates how to specify the stack base location:

For architectures where the stack grows down:

```
pStackMem = (char *) malloc (stackSize);

if (pStackMem != NULL)
    taskId = taskOpen ( ... , pStackMem + stackSize, stackSize, ... );
```

For architectures where the stack grows up:

```
pStackMem = (char *) malloc (stackSize);

if (pStackMem != NULL)
    taskId = taskOpen ( ... , pStackMem, stackSize, ... );
```

Please note that **malloc( )** is used in the above code fragment for illustrative purposes only since it's a well-known API. Typically, the stack memory would be obtained by some other mechanism.

It is assumed that if the caller passes a non-**NULL** pointer as *pStackBase*, it is valid. No validity check for this parameter is done here.

stackSize
The size in bytes of the execution stack area. If **NULL** pointer is specified as *pStackBase* and a negative value is specified for this parameter, the API returns **ERROR** considering it an illegal stack size. However, the API does not check against illegal stack size, if a non-**NULL** pointer is specified as *pStackBase*, since it is assumed that the user has allocated the stack memory with a valid stack size, before calling this API.

Every byte of the stack is filled with 0xee (unless the **VX_NO_STACK_FILL** option is specifed or the global kernel configuration parameter **VX_GLOBAL_NO_STACK_FILL** is set to **TRUE**) for the **checkStack( )** kernel shell facility.

context
Context value assigned to the created task. This value is not actually used by VxWorks. Instead, the context value can be used by OS extensions to implement object permissions, for example.

entryPt
The entry point is the address of the **main** routine of the task. The routine is called once the C environment has been set up. The specified routine is called with the ten arguments *arg1* to *arg10*. Should the specified **main** routine return, a call to **exit( )** is automatically made.

It is assumed that the caller passes a valid function pointer as *entryPt*. No validity check for this parameter is done here.

To delete a task created via the **taskOpen( )** API, **taskDelete( )** must be called. A call to **taskClose( )** will not perform the task deletion.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The task ID, or **NULL** if unsuccessful.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to spawn the task.

S_taskLib_ILLEGAL_PRIORITY
A priority outside the range 0 to 255 was specified.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to specify a location for the stack (not supported in TAR).

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid.

S_objLib_OBJ_NOT_FOUND
The **OM_CREATE** flag was not set in the *mode* argument and a task matching *name* was not found.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **taskOpen**, **taskSpawn( )**, **taskCreate( )**, **taskActivate( )**, **taskClose( )**, the VxWorks programmer guides.

taskOpenInit( )

NAME **taskOpenInit( )** – initialize the task open facility

SYNOPSIS
```
void taskOpenInit (void)
```

DESCRIPTION This routine links the task creation routine with the open facility into the VxWorks system. It is called automatically when the task facility is configured into VxWorks by either defining **INCLUDE_OBJ_OPEN** and **INCLUDE_TASK_CREATE_DELETE** in **config.h** or selecting **INCLUDE_OBJ_OPEN** and **INCLUDE_TASK_CREATE_DELETE** in the project facility.

RETURNS N/A

ERRNO N/A

SEE ALSO **taskOpen**

taskOptionsGet( )

NAME

taskOptionsGet( ) – examine task options

SYNOPSIS

```
STATUS taskOptionsGet
    (
    int   tid,      /* task ID */
    int * pOptions  /* task's options */
    )
```

DESCRIPTION

This routine gets the current execution options of the specified task. The option bits returned by this routine indicate the following modes:

VX_FP_TASK
execute with floating-point coprocessor support.

VX_PRIVATE_ENV
include private environment support (see **envLib**).

VX_NO_STACK_FILL
do not fill the stack for use by **checkstack( )**.

VX_UNBREAKABLE
do not allow breakpoint debugging.

For definitions, see **taskLib.h**.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNOS

N/A

SEE ALSO

taskInfo, **taskOptionsSet( )**

taskOptionsSet( )

NAME

taskOptionsSet( ) – change task options

SYNOPSIS

```
STATUS taskOptionsSet
    (
    int tid,        /* task ID */
    int mask,       /* bit mask of option bits to unset */
    int newOptions  /* bit mask of option bits to set */
    )
```

DESCRIPTION

This routine changes the execution options of a task. The only option that can be changed after a task has been created is:

VX_UNBREAKABLE
do not allow breakpoint debugging.

For definitions, see **taskLib.h**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNOS N/A

SEE ALSO **taskInfo**, **taskOptionsGet( )**

taskPriNormalGet( )

NAME **taskPriNormalGet( )** – get the normal priority of the task

SYNOPSIS

```
STATUS taskPriNormalGet
    (
    int  tid,        /* task ID */
    int* pPriNormal  /* where to return priority */
    )
```

DESCRIPTION This routine gets the normal priority of the specified task, which is the priority assigned at task creation time or subsequently assigned using **taskPrioritySet( )**. A task executes at its normal assigned priority unless priority inheritance has occurred.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNO N/A

SEE ALSO **taskInfo**, **taskSpawn( )**, **taskCreate( )**, **taskPrioritySet( )**

taskPriorityGet( )

NAME **taskPriorityGet( )** – examine the priority of a task

SYNOPSIS

```
STATUS taskPriorityGet
    (
    int   tid,       /* task ID */
    int * pPriority  /* return priority here */
    )
```

DESCRIPTION This routine determines the current priority of a specified task. The current priority is copied to the integer pointed to by *pPriority*.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO taskstack remains executablestack remains executable **skLib**, **taskPrioritySet( )**

taskPrioritySet( )

NAME **taskPrioritySet( )** – change the priority of a task

SYNOPSIS

```
STATUS taskPrioritySet
    (
    int tid,         /* task ID */
    int newPriority  /* new priority */
    )
```

DESCRIPTION This routine changes a task's priority to a specified priority. Priorities range from 0, the highest priority, to 255, the lowest priority.

A request to lower the priority of a task that has acquired a priority inversion safe mutex semaphore will not take immediate effect. To prevent a priority inversion situation, the requested lower priority will take effect, in general, only after the task relinquishes all priority inversion safe mutex semaphores.

A request to raise the priority of a task will take immediate effect.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNO **S_taskLib_ILLEGAL_PRIORITY**

S_objLib_OBJ_ID_ERROR

SEE ALSO **taskLib**, **taskPriorityGet( )**

taskRaise( )

NAME **taskRaise( )** – send a signal to the caller's task

SYNOPSIS

```
int taskRaise
    (
    int signo
    )
```

DESCRIPTION This routine sends the signal *signo* to the task invoking the call. This function is currently aliased to **raise( )**, and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

RETURNS **OK** (0), or **ERROR** (-1) if the signal number or task ID is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **raise( )**

taskRegsGet( )

NAME **taskRegsGet( )** – get a task's registers from the TCB

SYNOPSIS

```
STATUS taskRegsGet
    (
    int       tid,   /* task ID */
    REG_SET * pRegs  /* put register contents here */
    )
```

DESCRIPTION This routine gathers task information kept in the TCB. It copies the contents of the task's registers to the register structure *pRegs*.

NOTE This routine only works well if the task is known to be in a stable, non-executing state. Self-examination, for instance, is not advisable, as results are unpredictable.

SMP CONSIDERATIONS

Because of the concurrent execution environment of VxWorks SMP the specified task **must** explicitely be put in a non-executing state before calling this routine.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNOS N/A

SEE ALSO **taskInfo**, **taskSuspend( )**, **taskRegsSet( )**

taskRegsSet( )

NAME **taskRegsSet( )** – set a task's registers

SYNOPSIS

```
STATUS taskRegsSet
    (
    int       tid,    /* task ID */
    REG_SET * pRegs   /* get register contents from here */
    )
```

DESCRIPTION This routine loads a specified register set *pRegs* into a specified task's TCB.

NOTE This routine only works well if the task is known not to be in the ready state. Suspending the task before changing the register set is recommended.

SMP CONSIDERATIONS

Because of the concurrent execution environment of VxWorks SMP the specified task **must** explicitely be put in a non-executing state before calling this routine.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNOS N/A

SEE ALSO **taskInfo**, **taskSuspend( )**, **taskRegsGet( )**

taskRegsShow( )

NAME **taskRegsShow( )** – display the contents of a task's registers

SYNOPSIS

```
void taskRegsShow
    (
    int tid  /* task ID */
    )
```

DESCRIPTION This routine displays the register contents of a specified task on standard output.

EXAMPLE The following example displays the register of the shell task (PowerPC family):

```
-> taskRegsShow (taskNameToId ("tShell0"))

r0          = 0x00000000   sp          = 0x00c30ae0   r2          = 0x00000000
r3          = 0x00000000   r4          = 0x00000000   r5          = 0x00000000
r6          = 0x00000000   r7          = 0x00000000   r8          = 0x00000000
r9          = 0x00000000   r10         = 0x00000000   r11         = 0x00000000
r12         = 0x00000000   r13         = 0x00000000   r14         = 0x00000000
r15         = 0x00000000   r16         = 0x00000000   r17         = 0x00000000
r18         = 0x00000000   r19         = 0x00c1c02c   r20         = 0x02f93b20
r21         = 0x02f93c22   r22         = 0x0000005f   r23         = 0x00000000
r24         = 0x02f93e24   r25         = 0x00000003   r26         = 0x00373f80
r27         = 0x00c30b88   r28         = 0x00348494   r29         = 0xffffffff
r30         = 0x007877a4   r31         = 0x0200b030   msr         = 0x0200b030
lr          = 0x00000000   ctr         = 0x00000000   pc          = 0x0025d66c
cr          = 0x20000480   xer         = 0x00000000   pgTblPtr    = 0x00745000
scSrTblPtr = 0x007841f4   srTblPtr    = 0x007841f4
value = 1 = 0x1
```

RETURNS N/A

ERRNO N/A

SEE ALSO **taskShow**

taskRestart( )

NAME **taskRestart( )** – restart a task

SYNOPSIS

```
STATUS taskRestart
    (
    int tid  /* task ID of task to restart */
    )
```

DESCRIPTION

This routine "restarts" a task. The task is first terminated, and then reinitialized with the same ID, priority, options, original entry point, stack size, and parameters it had when it was terminated. Self-restarting of a calling task is performed by a newly spawned "tRestart" task. The shell utilizes this routine to restart itself when aborted.

NOTE

If the task has modified any of its start-up parameters, the restarted task will start with the changed values.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

WARNING

Restarting user mode tasks is not supported from kernel space, and may have unpredictable behavior. User mode tasks may be restarted from within an RTP (except for the initial task).

RETURNS

OK, or **ERROR** if the task ID is invalid or the task could not be restarted.

ERRNO

S_intLib_NOT_ISR_CALLABLE

S_objLib_OBJ_DELETED

S_objLib_OBJ_UNAVAILABLE

S_objLib_OBJ_ID_ERROR

S_smObjLib_NOT_INITIALIZED

S_memLib_NOT_ENOUGH_MEMORY

S_memLib_BLOCK_ERROR

S_taskLib_ILLEGAL_PRIORITY

SEE ALSO

taskLib

taskResume( )

NAME

taskResume( ) – resume a task

SYNOPSIS

```
STATUS taskResume
    (
    int tid  /* task ID of task to resume */
    )
```

DESCRIPTION

This routine resumes a specified task. Suspension is cleared, and the task operates in the remaining state.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task cannot be resumed.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskRotate( )

NAME **taskRotate( )** – rotate ready queue for a given task priority

SYNOPSIS

```
STATUS taskRotate
    (
    int priority  /* VX_TASK_PRIORITY_MIN to VX_TASK_PRIORITY_MAX */
    )
```

DESCRIPTION This routine rotates the ready queue of tasks that are ready to run for the priority specified by the *priority* parameter. In the special case that *priority* is set to **TASK_PRIORITY_SELF**, the ready queue for the caller's normal (spawned) priority is rotated. If no tasks are ready, or only one task is ready at the specified priority, no action is taken and this routine returns **OK** and leaves errno unchanged.

SMP CONSIDERATIONS

This routine is not supported for SMP. This routine will return **ERROR** if called from a SMP system.

NOTE The ITRON API **rot_rdq( )** can be implemented using the following macro.

#define rot_rdq(p) taskRotate (p == 0 ? **TASK_PRIORITY_SELF** : p)

or using the following function definition.

```
STATUS rot_rdq
    (
    UINT priority
    )
    {
    return taskRotate ( priority == 0 ? TASK_PRIORITY_SELF: priority);
   }
```

RETURNS

OK, or **ERROR**

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine is not callable from an ISR.

S_taskLib_ILLEGAL_PRIORITY
Priority specified is not within **VX_TASK_PRIORITY_MIN** to **VX_TASK_PRIORITY_MAX**.

SEE ALSO

taskRotate

taskSRInit( )

NAME

taskSRInit( ) – initialize the default task status register (MIPS)

SYNOPSIS

```
ULONG taskSRInit
    (
    ULONG newSRValue  /* new default task status register  */
    )
```

DESCRIPTION

This routine sets the default status register for system-wide tasks. All tasks will be spawned with the status register set to this value; thus, it must be called before **kernelInit( )**.

RETURNS

The previous value of the default status register.

ERRNO

Not Available

SEE ALSO

taskArchLib

taskSRSet( )

NAME

taskSRSet( ) – set the task status register (MC680x0, MIPS, x86)

SYNOPSIS

```
STATUS taskSRSet
    (
    int    tid,  /* task ID */
    UINT16 sr    /* new SR  */
    )
```

DESCRIPTION This routine sets the status register of a task that is not running (i.e., the TCB must not be that of the calling task). Debugging facilities use this routine to set the trace bit in the status register of a task that is being single-stepped.

x86:
The second parameter represents EFLAGS register and the size is 32 bit.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNO Not Available

SEE ALSO **taskArchLib**

taskSafe( )

NAME **taskSafe( )** – make the calling task safe from deletion

SYNOPSIS

```
STATUS taskSafe (void)
```

DESCRIPTION This routine protects the calling task from deletion. Tasks that attempt to delete a protected task will block until the task is made unsafe, using **taskUnsafe( )**. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The **taskSafe( )** primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost **taskUnsafe( )** is executed.

RETURNS **OK**.

ERRNO N/A

SEE ALSO **taskLib**, **taskUnsafe( )**, the VxWorks programmer guides.

taskShow( )

NAME **taskShow( )** – display task information from TCBs

SYNOPSIS

```
STATUS taskShow
    (
    int tid,   /* task ID */
    int level  /* 0 = summary, 1 = details, 2 = all tasks */
    )
```

DESCRIPTION

This routine displays the contents of a task control block (TCB) for a specified task. If *level* is 1, it also displays task options and registers. If *level* is 2, it displays all tasks in sorted order of the number of tasks are less than 500. If more than 500 tasks are in the system, *level* equal to 2 will display all tasks in the system unsorted and in the order they are created.

The TCB display contains the following fields:

Field	Meaning
NAME	Task name (truncated, if ending with a > character)
ENTRY	Symbol name or address where task began execution
TID	Task ID
PRI	Priority
STATUS	Task status, as formatted by **taskStatusString( )**
PC	Program counter
SP	Stack pointer
ERRNO	Most recent error code for this task
DELAY	If task is delayed, number of clock ticks remaining in delay (0 otherwise)
CPU #	For SMP systems, CPU index the task is running on ("-" otherwise)

Stack and register information for the specified task are also displayed.

SMP CONSIDERATIONS

Specifying a *level* of 2 will display a "CPU #" column instead of "DELAY". Specifying a *level* of 3 will result in the same output format as uniprocessor VxWorks, i.e. the "DELAY" column will be displayed.

EXAMPLE

The following example shows the TCB contents for the network task (PowerPC family):

```
UP version:

-> taskShow tNetTask, 1

  NAME         ENTRY       TID     PRI   STATUS      PC       SP      ERRNO
DELAY
----------  ------------ -------- --- ---------- -------- -------- -------
-----
tNetTask    netTask      7b3c50    50 PEND         25d66c   b056d0       0
0

task stack: base 0xb057a0   end 0xb03090   size 10000  high 1408   margin
8592
exc. stack: base 0xb067a0   end 0xb057b0   start 0xb067b0
exc. stack: size 4080       high 272       margin 3808

proc id: 0x36ffa8 ((null))
options: 0x9007
VX_SUPERVISOR_MODE  VX_UNBREAKABLE      VX_DEALLOC_STACK    VX_DEALLOC_TCB
```

```
VX_DEALLOC_EXC_STACK


VxWorks Events
--------------
Events Pended on    : Not Pended
Received Events     : 0x0
Options             : N/A

r0         = 0x00000000   sp         = 0x00b056d0   r2         = 0x00000000
r3         = 0x00000000   r4         = 0x00000000   r5         = 0x00000000
r6         = 0x00000000   r7         = 0x00000000   r8         = 0x00000000
r9         = 0x00000000   r10        = 0x00000000   r11        = 0x00000000
r12        = 0x00000000   r13        = 0x00000000   r14        = 0x00000000
r15        = 0x00000000   r16        = 0x00000000   r17        = 0x00000000
r18        = 0x00000000   r19        = 0x00000000   r20        = 0x00000000
r21        = 0x00000000   r22        = 0x00000000   r23        = 0x00000000
r24        = 0x00000000   r25        = 0x00000000   r26        = 0x00371adc
r27        = 0x0000b030   r28        = 0x00348494   r29        = 0xffffffff
r30        = 0x00371a40   r31        = 0x0000b030   msr        = 0x0000b030
lr         = 0x00000000   ctr        = 0x00000000   pc         = 0x0025d66c
cr         = 0x20000080   xer        = 0x00000000   pgTblPtr   = 0x00745000
scSrTblPtr = 0x007841f4   srTblPtr   = 0x007841f4
coprocTaskShow:  TaskId 0x7b3c50 has no coprocessors selected
value = 0 = 0x0

SMP version (shows TCB contents for the log task):

-> taskShow tLogTask, 1

  NAME         ENTRY        TID     PRI   STATUS       PC        SP     ERRNO
DELAY
----------  ------------ -------- --- ---------- -------- -------- -------
-----
tLogTask    logTask        2ce5a0   0 PEND         1ecad8   2ce480       0
0

task affinity: 0x0          task cpuIndex: -1 (Task Not Running)

task stack: base 0x2ce5a0    end 0x2cd210    size 5008   high 384    margin
4624
exc. stack: base 0x2cf7d0    end 0x2ce840    start 0x2cf840
exc. stack: size 3984        high 0          margin 3984

proc id: 0x245910 ((null))
options: 0x9003
VX_SUPERVISOR_MODE  VX_UNBREAKABLE      VX_DEALLOC_TCB
VX_DEALLOC_EXC_STACK

VxWorks Events
--------------
Events Pended on    : Not Pended
Received Events     : 0x0
Options             : N/A

r0         = 0x00000000   sp         = 0x002ce480   r2         = 0x00000000
```

```
r3         = 0x00000000  r4         = 0x00000000  r5         = 0x00000000
r6         = 0x00000000  r7         = 0x00000000  r8         = 0x00000000
r9         = 0x00000000  r10        = 0x00000000  r11        = 0x00000000
r12        = 0x00000000  r13        = 0x00000000  r14        = 0x00000000
r15        = 0x00000000  r16        = 0x00000000  r17        = 0x00000000
r18        = 0x00000000  r19        = 0x00000000  r20        = 0x00000000
r21        = 0x00000000  r22        = 0x00000000  r23        = 0x00000000
r24        = 0x00000000  r25        = 0x002ce4b8  r26        = 0x00000000
r27        = 0xffffffff  r28        = 0x00000001  r29        = 0x002cc830
r30        = 0x002cc7f0  r31        = 0xffffffff  msr        = 0x0000b030
lr         = 0x00000000  ctr        = 0x00000000  pc         = 0x001ecad8
cr         = 0x20000000  xer        = 0x00000000  pgTblPtr   = 0x00279000
scSrTblPtr = 0x00278154  srTblPtr   = 0x00278154
coprocTaskShow:  TaskId 0x2ce5a0 has no coprocessors selected
value = 0 = 0x0

SMP level 3 taskShow example:

-> taskShow 0, 3

  NAME         ENTRY        TID     PRI   STATUS       PC        SP      ERRNO
DELAY
----------  ------------ -------- --- ---------- -------- -------- -------
-----
tExcTask    192c6c        26ce00   0 PEND         1ef424   26ef80       0
0
tJobTask    193bc4        2cb440   0 PEND         1ef424   2cb380       0
0
tLogTask    logTask       2ce5a0   0 PEND         1ecad8   2ce480       0
0
tNbioLog    19507c        2d1e20   0 PEND         1ef424   2d1d10       0
0
tShell0     shellTask     2e4130   1 READY        1f8164   2e23e0       0
0
miiBusMoni> 140fc8        2c3010 254 DELAY        1f5844   2c2f80       0
50
tIdleTask0  idleTaskEntr   272c30 287 READY        1eeda8   272bb0       0
0
tIdleTask1  idleTaskEntr   276250 287 READY        1eedb4   2761d0       0
0
value = 0 = 0x0
```

If the specified task uses coprocessors, such as a floating point coprocessor, this routine will also display the registers for the corresponding coprocessor.

If this routine is called with the current task as the argument, register information will not be displayed. Use **taskRegsShow( )** to display the register information for the calling task instead.

RETURNS N/A

ERRNO N/A

SEE ALSO **taskShow**, **taskStatusString( )**, **windsh**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

taskShowInit( )

NAME **taskShowInit( )** – initialize the task show routine facility

SYNOPSIS

```
void taskShowInit (void)
```

DESCRIPTION This routine links the task show routines into the VxWorks system. It is called automatically when the task 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 project facility, select **INCLUDE_TASK_SHOW**.

RETURNS N/A

ERRNO N/A

SEE ALSO **taskShow**

taskSigqueue( )

NAME **taskSigqueue( )** – send a queued signal to a task

SYNOPSIS

```
int taskSigqueue
    (
    int                 tid,
    int                 signo,
    const union sigval  value
    )
```

DESCRIPTION The function **sigqueue( )** sends the signal specified by *signo* with the signal-parameter value specified by *value* to the process specified by *tid*.

This function is currently aliased to **sigqueue( )**, and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**

EAGAIN

SEE ALSO **sigLib**, **sigqueue( )**

taskSpareFieldGet( )

NAME **taskSpareFieldGet( )** – get the spare field of a TCB

SYNOPSIS

```
int taskSpareFieldGet
    (
    int       tid,          /* task ID */
    SPARE_NUM numAllotted  /* spare field to get */
    )
```

DESCRIPTION This routine gets the value of a spare field. The spare field was gotten by calling **taskSpareNumAllot( )** to get the available spare field to use.

RETURNS value of the spare field, or **ERROR** if task or numAllotted is invalid

ERRNO N/A

SEE ALSO **taskUtilLib**, **taskSpareNumAllot( )**, **taskSpareFieldSet( )**

taskSpareFieldSet( )

NAME **taskSpareFieldSet( )** – set the spare field of a TCB

SYNOPSIS

```
STATUS taskSpareFieldSet
    (
    int       tid,           /* task ID */
    SPARE_NUM numAllotted,   /* spare field to set */
    int       value          /* value to set */
    )
```

DESCRIPTION This routine sets the value of a spare field. The spare field is gotten by calling **taskSpareNumAllot( )** to get the available spare field to use.

An example:

```
int spareNum;
taskSpareNumAllot (t1, &spareNum);
if (spareNum != ERROR)
    taskSpareFieldSet (t1, spareNum, 0x12345678);
```

RETURNS **OK**, or **ERROR** if task is invalid or numAllotted is invalid

ERRNOS N/A

SEE ALSO **taskUtilLib**, **taskSpareFieldGet( )**, **taskSpareNumAllot( )**

taskSpareNumAllot( )

NAME **taskSpareNumAllot( )** – Allocate the first available spare field in the TCB

SYNOPSIS

```
void taskSpareNumAllot
    (
    int         tid,         /* task ID */
    SPARE_NUM * numAllotted  /* where to return SPARE_NUM */
    )
```

DESCRIPTION This routine allots the first available spare field in the TCB. Once a spare field is allotted, this same number is used to reference the same spare field on all WIND_TCBs in the system. In other words, two different tasks, t1 and t2, will get different **SPARE_NUM** values when called.

Once a field has been allotted, the field is reserved for the life of the system. A field may not be un-allotted or unreserved.

RETURNS **SPARE_NUM**, or **ERROR** if fields are not available

ERRNOS N/A

SEE ALSO **taskUtilLib**, **taskSpareFieldGet( )**, **taskSpareFieldSet( )**

taskSpawn( )

NAME

taskSpawn( ) – spawn a task

SYNOPSIS

```
int taskSpawn
    (
    char *  name,       /* name of new task (stored at pStackBase) */
    int     priority,   /* priority of new task */
    int     options,    /* task option word */
    int     stackSize,  /* size (bytes) of stack needed plus name */
    FUNCPTR entryPt,    /* entry point of new task */
    int     arg1,       /* 1st of 10 req'd args to pass to entryPt */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9,
    int     arg10
    )
```

DESCRIPTION

This routine creates and activates a new task with a specified priority and options and returns a system-assigned ID. See **taskInit( )** and **taskActivate( )** for the building blocks of this routine.

A task may be assigned a name as a debugging aid. This name will appear in displays generated by various system information facilities such as **i( )**. The name may be of arbitrary length and content, but the current VxWorks convention is to limit task names to ten characters and prefix them with a "t". If *name* is specified as **NULL**, an ASCII name will be assigned to the task of the form "t*n*" where *n* is an integer which increments as new tasks are spawned.

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 in dynamic and one may change an existing task's priority with **taskPrioritySet( )**.

The only resource allocated to a spawned task is a stack of a specified size *stackSize*, which is allocated from the system memory partition. Stack size should be an even integer. A task control block (TCB) is carved from the stack, as well as any memory required by the task name. The remaining memory is the task's stack and every byte is filled with the value 0xEE (unless the **VX_NO_STACK_FILL** option is specifed or the global kernel configuration parameter **VX_GLOBAL_NO_STACK_FILL** is set to **TRUE**) for the **checkStack( )** facility. See the manual entry for **checkStack( )** for stack-size checking aids.

The entry address *entryPt* is the address of the "main" routine of the task. The routine will be called once the C environment has been set up. The specified routine will be called with the ten given arguments. Should the specified main routine return, a call to **exit( )** will automatically be made.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes:

VX_FP_TASK
: execute with floating-point coprocessor support. A task which performs floating point operations or calls any functions which either return or take a floating point value as arguments must be created with this option. Some routines perform floating point operations internally. The VxWorks documentation for these clearly state the need to use the **VX_FP_TASK** option.

VX_ALTIVEC_TASK
: execute with Altivec support (PowerPC only)

VX_SPE_TASK
: execute with SPE support (PowerPC only)

VX_DSP_TASK
: execute with DSP support (SuperH only)

VX_PRIVATE_ENV
: include private environment support (see **envLib**).

VX_NO_STACK_FILL
: do not fill the stack for use by **checkStack( )**.

VX_UNBREAKABLE
: do not allow breakpoint debugging.

VX_NO_STACK_PROTECT
: do not provide stack protection: no overflow or underflow detection, stack remains executable.

The option bits **VX_DEALLOC_STACK** and **VX_DEALLOC_EXC_STACK** are not options available for the **taskSpawn( )** API. **taskSpawn( )** internally sets these option bits by default depending on the configuration of the system. Specifying these options to **taskSpawn( )** results in an **ERROR** and the ERRNO, **S_taskLib_ILLEGAL_OPTIONS**, will be returned. See the definitions in **taskLib.h**.

It is assumed that the caller passes a valid function pointer as *entryPt* while calling this API. No validity check for this parameter is done here.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

The task ID, or **ERROR** if memory is insufficient or the task cannot be created.

ERRNO

S_intLib_NOT_ISR_CALLABLE
: Routine is not callable from an ISR.

S_objLib_OBJ_ID_ERROR

S_smObjLib_NOT_INITIALIZED

S_memLib_NOT_ENOUGH_MEMORY
Out of memory for allocation of stack or TCB.

S_memLib_BLOCK_ERROR

S_taskLib_ILLEGAL_PRIORITY
Priority specified is not within 0-255.

S_taskLib_ILLEGAL_OPTIONS
The following illegal options were set: **VX_DEALLOC_STACK**, **VX_DEALLOC_EXC_STACK**, or **VX_DEALLOC_TCB** for **taskSpawn( )**.

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified

SEE ALSO **taskLib**, **taskInit( )**, **taskActivate( )**, **sp( )**, the VxWorks programmer guides.

taskStackAllot( )

NAME **taskStackAllot( )** – allot memory from a task's exception stack

SYNOPSIS

```
void * taskStackAllot
    (
    int      tid,    /* task whose stack will be allotted from */
    unsigned nBytes  /* number of bytes to allot */
    )
```

DESCRIPTION This routine allots the specified amount of memory from the start of the exception stack of the task specified by *tid*. This is a non-blocking operation meant to be used by task create hooks that need to allocate small amounts of memory on a per-task basis. Since the memory is carved from the exception stack calling this routine essentially causes the amount of stack space available for execution to be reduced. Hence the requirement to only allocate small amounts of memory. The exception stack size of a kernel task cannot be modified. The exception stack size of tasks that run in real-time processes is controlled by the **USER_TASK_EXC_STACK_SIZE** configuration parameter of the **INCLUDE_KERNEL** component.

It is an error condition for a task to call this routine to attempt to allocate memory from its own exception stack. Attempting to do this results in this routine returning **NULL**. Attempting to allocate memory from the exception stack of a task that has started execution has undefined results. This routine is meant to only ever be called before a task initially starts executing.

The memory allocated with this routine cannot be added back to the task's exception stack. It will be reclaimed as part of the reclamation of the exception stack when the task is deleted.

Note that an exception stack underrun will overwrite the allotments made from this routine because all portions are carved from the start of the exception stack.

This routine returns **NULL** if the requested size exceeds available stack memory.

RETURNS pointer to block, or **NULL** if unsuccessful.

ERRNO N/A

SEE ALSO **taskLib**, **taskCreateHookAdd( )**

taskStatusString( )

NAME **taskStatusString( )** – get a task's status as a string

SYNOPSIS

```
STATUS taskStatusString
    (
    int    tid,     /* task to get string for */
    char * pString  /* where to return string */
    )
```

DESCRIPTION This routine deciphers the WIND task status word in the TCB for a specified task, and copies the appropriate string to *pString*.

The formatted string is one of the following:

String	Meaning
READY	Task is not waiting for any resource other than the CPU.
PEND	Task is blocked due to the unavailability of some resource.
DELAY	Task is asleep for some duration.
SUSPEND	Task is unavailable for execution (but not delayed, or pended).
STOP	Task is stopped by the debugger.
DELAY+S	Task is both delayed and suspended.
PEND+S	Task is both pended and suspended.
PEND+T	Task is pended with a timeout.
STOP+P	Task is both pended and stopped by the debugger.
STOP+S	Task is both stopped by the debugger and suspended.
STOP+T	Task is both delayed and stopped by the debugger.
PEND+S+T	Task is pended with a timeout, and also suspended.
STOP+P+S	Task is pended, suspended, and also stopped by the debugger.
STOP+P+T	Task is pended with a timeout, and also stopped by the debugger.
STOP+S+T	Task is suspended with a timeout, and also stopped by the debugger.

String	Meaning
ST+P+S+T	Task is pended with a timeout, suspended, and stopped by the debugger.
...+I	Task has inherited priority (+I may be appended to any string above).
DEAD	Task no longer exists.

EXAMPLE

```
    -> taskStatusString (taskNameToId ("tShell0"), xx=malloc (10))
    new symbol "xx" added to symbol table.
    value = 0 = 0x0
    -> printf ("shell status = <%s>\n", xx)
    shell status = <READY>
    value = 2 = 0x2
```

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNO

N/A

SEE ALSO

taskShow

taskSuspend( )

NAME

taskSuspend( ) – suspend a task

SYNOPSIS

```
STATUS taskSuspend
    (
    int tid  /* task ID of task to suspend */
    )
```

DESCRIPTION

This routine suspends a specified task. A task ID of zero results in the suspension of the calling task. Suspension is additive, thus tasks can be delayed and suspended, or pended and suspended. Suspended, delayed tasks whose delays expire remain suspended. Likewise, suspended, pended tasks that unblock remain suspended only.

Care should be taken with asynchronous use of this facility. The specified task is suspended regardless of its current state. The task could, for instance, have mutual exclusion to some system resource, such as the network or system memory partition. If suspended during such a time, the facilities engaged are unavailable, and the situation often ends in deadlock.

This facility should be rejected as a synchronization mechanism in favor of the more general semaphore facility.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if the task cannot be suspended.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskSwitchHookAdd( )

NAME **taskSwitchHookAdd( )** – add a routine to be called at every task switch

SYNOPSIS

```
STATUS taskSwitchHookAdd
    (
    FUNCPTR switchHook  /* routine to be called at every task switch */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called at every task switch. The routine should be declared as follows:

```
    void switchHook
        (
        WIND_TCB *pOldTcb,  /* pointer to old task's WIND_TCB */
        WIND_TCB *pNewTcb   /* pointer to new task's WIND_TCB */
        )
```

NOTE User-installed switch hooks are called within the kernel context. Therefore, switch hooks do not have access to all VxWorks facilities. The following routines can be called from within a task switch hook:

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

RETURNS **OK**, or **ERROR** if the table of task switch routines is full.

ERRNO Not Available

SEE ALSO **taskHookLib**, **taskSwitchHookDelete( )**

taskSwitchHookDelete( )

NAME

taskSwitchHookDelete( ) – delete a previously added task switch routine

SYNOPSIS

```
STATUS taskSwitchHookDelete
    (
    FUNCPTR switchHook  /* routine to be deleted from list */
    )
```

DESCRIPTION

This routine removes the specified routine from the list of routines to be called at each task switch.

RETURNS

OK, or **ERROR** if the routine is not in the table of task switch routines.

ERRNO

Not Available

SEE ALSO

taskHookLib, **taskSwitchHookAdd( )**

taskSwitchHookShow( )

NAME

taskSwitchHookShow( ) – show the list of task switch routines

SYNOPSIS

```
void taskSwitchHookShow (void)
```

DESCRIPTION

This routine shows all the switch routines installed in the task switch hook table, in the order in which they were installed.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

taskHookShow, **taskSwitchHookAdd( )**

taskTcb( )

NAME

taskTcb( ) – get the task control block for a task ID

SYNOPSIS

```
WIND_TCB * taskTcb
```

```
    (
    int tid  /* task ID */
    )
```

DESCRIPTION This routine returns a pointer to the task control block (**WIND_TCB**) for a specified task. Although all task state information is contained in the TCB, users must not modify it directly. To change registers, for instance, use **taskRegsSet( )** and **taskRegsGet( )**.

RETURNS A pointer to a **WIND_TCB**, or **NULL** if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskUnlink( )

NAME **taskUnlink( )** – unlink a task

SYNOPSIS

```
STATUS taskUnlink
    (
    const char * name  /* name of task to unlink */
    )
```

DESCRIPTION This routine removes a task from its name space. The use of this routine on private tasks, which support duplicate names, is not recomended. After a task is unlinked, subsequent calls to **taskOpen( )** using *name* will not be able to find the task, even if it has not been deleted yet. Instead, a new task could be created if **taskOpen( )** is called with the **OM_CREATE** flag.

This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL** or empty.

S_objLib_OBJ_NOT_FOUND
No task with *name* was found.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **taskOpen**, **taskOpen( )**, **taskClose( )**

taskUnlock( )

NAME **taskUnlock( )** – enable task rescheduling

SYNOPSIS

```
STATUS taskUnlock
    (
    void
    )
```

DESCRIPTION This routine decrements the preemption lock count. Typically this call is paired with **taskLock( )** and concludes a critical section of code. Preemption will not be unlocked until **taskUnlock( )** has been called as many times as **taskLock( )**. When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

The **taskUnlock( )** routine is not callable from interrupt service routines.

SMP CONSIDERATIONS
This API is not available in VxWorks SMP.

RETURNS **OK** or **ERROR**.

ERRNO **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **taskLib**, **taskLock( )**, **taskCpuUnlock( )**

taskUnsafe( )

NAME **taskUnsafe( )** – make the calling task unsafe from deletion

SYNOPSIS

```
STATUS taskUnsafe (void)
```

DESCRIPTION This routine removes the calling task's protection from deletion. Tasks that attempt to delete a protected task will block until the task is unsafe. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The **taskUnsafe( )** primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost **taskUnsafe( )** is executed.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS **OK**.

ERRNO N/A

SEE ALSO **taskLib**, **taskSafe( )**, the VxWorks programmer's guides

taskVarAdd( )

NAME **taskVarAdd( )** – add a task variable to a task

SYNOPSIS

```
STATUS taskVarAdd
    (
    int tid,   /* ID of task to have new variable */
    int *pVar  /* pointer to variable to be switched for task */
    )
```

DESCRIPTION This routine adds a specified variable *pVar* (4-byte memory location) to a specified task's context. After calling this routine, the variable will be private to the task. The task can access and modify the variable, but the modifications will not appear to other tasks, and other tasks' modifications to that variable will not affect the value seen by the task. This is accomplished by saving and restoring the variable's initial value each time a task switch occurs to or from the calling task.

This facility can be used when a routine is to be spawned repeatedly as several independent tasks. Although each task will have its own stack, and thus separate stack variables, they will all share the same static and global variables. To make a variable *not* shareable, the routine can call **taskVarAdd( )** to make a separate copy of the variable for each task, but all at the same physical address.

Note that task variables increase the task switch time to and from the tasks that own them. Therefore, it is desirable to limit the number of task variables that a task uses. One efficient way to use task variables is to have a single task variable that is a pointer to a dynamically allocated structure containing the task's private data.

EXAMPLE

Assume that three identical tasks were spawned with a routine called **operator( )**. All three use the structure **OP_GLOBAL** for all variables that are specific to a particular incarnation of the task. The following code fragment shows how this is set up:

```
OP_GLOBAL *opGlobal;  /* ptr to operator task's global variables */

void operator
    (
    int opNum         /* number of this operator task */
    )
    {
    if (taskVarAdd (0, (int *)&opGlobal) != OK)
        {
        printErr ("operator%d: can't taskVarAdd opGlobal\\n", opNum);
        taskSuspend (0);
        }

    if ((opGlobal = (OP_GLOBAL *) malloc (sizeof (OP_GLOBAL))) == NULL)
        {
        printErr ("operator%d: can't malloc opGlobal\\n", opNum);
        taskSuspend (0);
        }
    ...
    }
```

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Use *__thread* variables instead

RETURNS

OK, or **ERROR** if memory is insufficient for the task variable descriptor or semaphore.

ERRNOS

no errnos for this routine

SEE ALSO

taskVarLib, **taskVarDelete( )**, **taskVarGet( )**, **taskVarSet( )**

taskVarDelete( )

NAME

taskVarDelete( ) – remove a task variable from a task

SYNOPSIS

```
STATUS taskVarDelete
    (
    int tid,   /* ID of task whose variable is to be removed */
    int *pVar  /* pointer to task variable to be removed */
    )
```

DESCRIPTION

This routine removes a specified task variable, *pVar*, from the specified task's context. The private value of that variable is lost.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Use __*thread* variables instead.

RETURNS **OK**, or **ERROR** if the task variable does not exist for the specified task.

ERRNOS Possible errno values set by this routine are:

S_taskLib_TASK_VAR_NOT_FOUND - address specified in *pVar* is not a task variable for *tid*

SEE ALSO **taskVarLib**, **taskVarAdd( )**, **taskVarGet( )**, **taskVarSet( )**

taskVarGet( )

NAME **taskVarGet( )** – get the value of a task variable

SYNOPSIS

```
int taskVarGet
    (
    int tid,   /* ID of task whose task variable is to be retrieved */
    int *pVar  /* pointer to task variable */
    )
```

DESCRIPTION This routine returns the private value of a task variable for a specified task. The specified task is usually not the calling task, which can get its private value by directly accessing the variable. This routine is provided primarily for debugging purposes.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Use __*thread* variables instead.

RETURNS The private value of the task variable, or **ERROR** if the task is not found or it does not own the task variable.

ERRNOS Possible errno values set by this routine are:

S_taskLib_TASK_VAR_NOT_FOUND - address specified in *pVar* is not a task variable for *tid*

SEE ALSO **taskVarLib**, **taskVarAdd( )**, **taskVarDelete( )**, **taskVarSet( )**

taskVarInfo( )

NAME

taskVarInfo( ) – get a list of task variables of a task

SYNOPSIS

```
int taskVarInfo
    (
    int      tid,         /* ID of task whose task variable is to be set */
    TASK_VAR varList[],   /* array to hold task variable addresses */
    int      maxVars      /* maximum variables varList can accommodate */
    )
```

DESCRIPTION

This routine provides the calling task with a list of all of the task variables of a specified task. The unsorted array of task variables is copied to *varList*.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP.

RETURNS

The number of task variables in the list or **ERROR** if the specified task ID is not valid.

ERRNOS

no errnos for this routine

SEE ALSO

taskVarLib

taskVarInit( )

NAME

taskVarInit( ) – initialize the task variables facility

SYNOPSIS

```
STATUS taskVarInit (void)
```

DESCRIPTION

This routine initializes the task variables facility. It installs task switch and delete hooks used for implementing task variables. If **taskVarInit( )** is not called explicitly, **taskVarAdd( )** will call it automatically when the first task variable is added.

After the first invocation of this routine, subsequent invocations have no effect.

WARNING

Order dependencies in task delete hooks often involve task variables. If a facility uses task variables and has a task delete hook that expects to use those task variables, the facility's delete hook must run before the task variables' delete hook. Otherwise, the task variables will be deleted by the time the facility's delete hook runs.

VxWorks is careful to run the delete hooks in reverse of the order in which they were installed. Any facility that has a delete hook that will use task variables can guarantee proper ordering by calling **taskVarInit( )** before adding its own delete hook.

Note that this is not an issue in normal use of task variables. The issue only arises when adding another task delete hook that uses task variables.

Caution should also be taken when adding task variables from within create hooks. If the task variable package has not been installed via **taskVarInit( )**, the create hook attempts to create a create hook, and that may cause system failure. To avoid this situation, **taskVarInit( )** should be called during system initialization from the root task, **usrRoot( )**, in **usrConfig.c**.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP.

RETURNS

OK, or **ERROR** if the task switch/delete hooks could not be installed.

ERRNOS

no errnos for this routine

SEE ALSO

taskVarLib

taskVarSet( )

NAME

taskVarSet( ) – set the value of a task variable

SYNOPSIS

```
STATUS taskVarSet
    (
    int tid,     /* ID of task whose task variable is to be set */
    int *pVar,   /* pointer to task variable to be set for this task */
    int value    /* new value of task variable */
    )
```

DESCRIPTION

This routine sets the private value of the task variable for a specified task. The specified task is usually not the calling task, which can set its private value by directly modifying the variable. This routine is provided primarily for debugging purposes.

SMP CONSIDERATIONS

This routine is not available in VxWorks SMP. Use *__thread* variables instead.

RETURNS

OK, or **ERROR** if the task is not found or it does not own the task variable.

ERRNOS

Possible errno values set by this routine are:

S_taskLib_TASK_VAR_NOT_FOUND - address specified in *pVar* is not a task variable for *tid*

SEE ALSO

taskVarLib, **taskVarAdd( )**, **taskVarDelete( )**, **taskVarGet( )**

td( )

NAME

td( ) – delete a task

SYNOPSIS

```
void td
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION

This command deletes a specified task. It simply calls **taskDelete( )**.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

usrLib, **taskDelete( )**, the VxWorks programmer guides.

tffsDevCreate( )

NAME

tffsDevCreate( ) – create a TrueFFS block device suitable for use with dosFs

SYNOPSIS

```
BLK_DEV * tffsDevCreate
    (
    int tffsDriveNo,        /* TFFS drive number (0 - DRIVES-1) */
    int removableMediaFlag  /* 0 - nonremovable flash media */
    )
```

DESCRIPTION

This routine creates a TFFS block device on top of a flash device. It takes as arguments a drive number, determined from the order in which the socket components were registered, and a flag integer that indicates whether the medium is removable or not. A zero indicates a non removable medium. A one indicates a removable medium. If you intend to mount dosFs on this block device, you probably do not want to call **tffsDevCreate( )**, but should call **usrTffsConfig( )** instead. Internally, **usrTffsConfig( )** calls **tffsDevCreate( )** for you. It then does everything necessary (such as calling the **dosFsDevInit( )** routine) to mount dosFs on the just created block device.

RETURNS

BLK_DEV pointer, or **NULL** if it failed.

ERRNO

Not Available

SEE ALSO

tffsDrv

tffsDevFormat( )

NAME **tffsDevFormat( )** – format a flash device for use with TrueFFS

SYNOPSIS

```
STATUS tffsDevFormat
    (
    int tffsDriveNo,  /* TrueFFS drive number (0 - DRIVES-1) */
    int arg           /* pointer to tffsDevFormatParams structure */
    )
```

DESCRIPTION This routine formats a flash device for use with TrueFFS. It takes two parameters, a drive number and a pointer to a device format structure. This structure describes how the volume should be formatted. The structure is defined in **dosformt.h**. The drive number is assigned in the order that the socket component for the device was registered.

The format process marks each erase unit with an Erase Unit Header (EUH) and creates the physical and virtual Block Allocation Maps (BAM) for the device. The erase units reserved for the "boot-image" are skipped and the first EUH is placed at number (boot-image length - 1). To write to the boot-image region, call **tffsBootImagePut( )**.

WARNING If any of the erase units in the boot-image region contains an erase unit header from a previous format call (this can happen if you reformat a flash device specifying a larger boot region) TrueFFS fails to mount the device. To fix this problem, use **tffsRawio( )** to erase the problem erase units (thus removing the outdated EUH).

The macro **TFFS_STD_FORMAT_PARAMS** defines the default values used for formatting a flask disk device. If the second argument to this routine is zero, **tffsDevFormat( )** uses these default values.

RETURNS **OK**, or **ERROR** if it failed.

ERRNO Not Available

SEE ALSO **tffsDrv**

tffsDevOptionsSet( )

NAME **tffsDevOptionsSet( )** – set TrueFFS volume options

SYNOPSIS

```
STATUS tffsDevOptionsSet
    (
    TFFS_DEV * pTffsDev  /* pointer to device descriptor */
    )
```

DESCRIPTION This routine is intended to set various TrueFFS volume options. At present it only disables FAT monitoring. If VxWorks long file names are to be used with TrueFFS, FAT monitoring must be turned off.

RETURNS **OK**, or **ERROR** if it failed.

ERRNO Not Available

SEE ALSO **tffsDrv**

tffsDrv( )

NAME **tffsDrv( )** – initialize the TrueFFS system

SYNOPSIS

```
STATUS tffsDrv (void)
```

DESCRIPTION This routine sets up the structures, the global variables, and the mutual exclusion semaphore needed to manage TrueFFS. This call also registers socket component drivers for all the flash devices attached to your target.

Because **tffsDrv( )** is the call that initializes the TrueFFS system, this function must be called (exactly once) before calling any other TrueFFS utilities, such as **tffsDevFormat( )** or **tffsDevCreate( )**. Typically, the call to **tffsDrv( )** is handled for you automatically. If you defined **INCLUDE_TFFS** in your BSP's **config.h**, the call to **tffsDrv( )** is made from **usrRoot( )**. If your BSP's **config.h** defines **INCLUDE_PCMCIA**, the call to **tffsDrv( )** is made from **pccardTffsEnabler( )**.

RETURNS **OK**, or **ERROR** if it fails.

ERRNO Not Available

SEE ALSO **tffsDrv**

tffsDrvOptionsSet( )

NAME **tffsDrvOptionsSet( )** – set TrueFFS volume options

SYNOPSIS

```
STATUS tffsDrvOptionsSet
```

```
    (
    int tffsDriveNo  /* TFFS drive number (0 - DRIVES-1) */
    )
```

DESCRIPTION This routine is intended to set various TrueFFS volume options. At present it only disables FAT monitoring. If VxWorks long file names are to be used with TrueFFS, FAT monitoring must be turned off. If Datalite's Reliance file file system is to be used with TrueFFS, FAT monitoring must be turned off.

RETURNS **OK**, or **ERROR** if it failed.

ERRNO Not Available

SEE ALSO **tffsDrv**

tffsRawio( )

NAME **tffsRawio( )** – low level I/O access to flash components

SYNOPSIS

```
STATUS tffsRawio
    (
    int tffsDriveNo,  /* TrueFFS drive number (0 - DRIVES-1) */
    int functionNo,   /* TrueFFS function code */
    int arg0,         /* argument 0 */
    int arg1,         /* argument 1 */
    int arg2          /* argument 2 */
    )
```

DESCRIPTION Use the utilities provided by thisroutine with the utmost care. If you use these routines carelessly, you risk data loss as well as permanent physical damage to the flash device.

This routine is a gateway to a series of utilities (listed below). Functions such as **mkbootTffs( )** and **tffsBootImagePut( )** use these **tffsRawio( )** utilities to write boot sector information. The functions for physical read, write, and erase are made available with the intention that they be used on erase units allocated to the boot-image region by **tffsDevFormat( )**. Using these functions elsewhere could be dangerous.

The *arg0*, *arg1*, and *arg2* parameters to **tffsRawio( )** are interpreted differently depending on the function number you specify for *functionNo*. The drive number is determined by the order in which the socket components were registered.

Function Name	arg0	arg1	arg2
TFFS_GET_PHYSICAL_INFO	user buffer address	N/A	N/A

Function Name	arg0	arg1	arg2
TFFS_PHYSICAL_READ	address to read	byte count	user buffer address
TFFS_PHYSICAL_WRITE	address to write	byte count	user buffer address
TFFS_PHYSICAL_ERASE	first unit	number of units	N/A
TFFS_ABS_READ	sector number	number of sectors	user buffer address
TFFS_ABS_WRITE	sector number	number of sectors	user buffer address
TFFS_ABS_DELETE	sector number	number of sectors	N/A
TFFS_DEFRAGMENT_VOLUME	number of sectors	user buffer address	N/A

TFFS_GET_PHYSICAL_INFO
writes the flash type, erasable block size, and media size to the user buffer specified in *arg0*.

TFFS_PHYSICAL_READ
reads *arg1* bytes from *arg0* and writes them to the buffer specified by *arg2*.

TFFS_PHYSICAL_WRITE
copies *arg1* bytes from the *arg2* buffer and writes them to the flash memory location specified by *arg0*. This aborts if the volume is already mounted to prevent the versions of translation data in memory and in flash from going out of synchronization.

TFFS_PHYSICAL_ERASE
erases *arg1* erase units, starting at the erase unit specified in *arg0*. This aborts if the volume is already mounted to prevent the versions of translation data in memory and in flash from going out of synchronization.

TFFS_ABS_READ
reads *arg1* sectors, starting at sector *arg0*, and writes them to the user buffer specified in *arg2*.

TFFS_ABS_WRITE
takes data from the *arg2* user buffer and writes *arg1* sectors of it to the flash location starting at sector *arg0*.

TFFS_ABS_DELETE
deletes *arg1* sectors of data starting at sector *arg0*.

TFFS_DEFRAGMENT_VOLUME
calls the defragmentation routine with the minimum number of sectors to be reclaimed, *arg0*, and writes the actual number reclaimed in the user buffer by *arg1*. Calling this function through some low priority task will make writes more deterministic. No validation is done of the user specified address fields, so the functions assume they are writable. If the address is invalid, you could see bus errors or segmentation faults.

RETURNS **OK**, or **ERROR** if it failed.

ERRNO Not Available

SEE ALSO **tffsDrv**

ti( )

NAME **ti( )** – print complete information from a task's TCB

SYNOPSIS

```
void ti
    (
    int taskNameOrId  /* task name or task ID; 0 = use default */
    )
```

DESCRIPTION This command prints the task control block (TCB) contents, including registers, for a specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

The **ti( )** routine uses **taskShow( )**; see the documentation for **taskShow( )** for a description of the output format.

EXAMPLE The following shows the TCB contents for the shell task:

```
-> ti

  NAME       ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
--------- ----------- -------- --- ---------- -------- -------- ------- -----
tShell0   shellTask   60351ba8   1 READY      6015fe68 603508d0  ad0007     0

task stack: base 0x60351ba8  end 0x6033eba8  size 77824  high 14144  margin
63680
exc. stack: base 0x60354f18  end 0x60351fd8  start 0x60354fd8
exc. stack: size 12096  high 4600   margin 7496

proc id: 0x60187008 ((null))
options: 0x1001007
VX_SUPERVISOR_MODE  VX_UNBREAKABLE      VX_DEALLOC_STACK
VX_DEALLOC_EXC_STACK VX_FP_TASK

VxWorks Events
--------------
Events Pended on    : Not Pended
Received Events     : 0x0
Options             : N/A
value = 0 = 0x0
```

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **taskShow( )**, the VxWorks programmer guides.

tick64Get( )

NAME **tick64Get( )** – get the value of the kernel's tick counter as a 64 bit value

SYNOPSIS

```
UINT64 tick64Get (void)
```

DESCRIPTION This routine returns the current value of the 64 bit absolute tick counter. This value is set to zero at startup, incremented by **tickAnnounce( )**, and can be changed using **tickSet( )** or **tick64Set( )**.

SMP CONSIDERATIONS

In SMP configuration this API is spinLock restricted meaning that calling this API while holding a spinLock is not allowed. Not comforming to this restriction will potentially lead to a live-lock scenerio.

RETURNS The most recent **tickSet( )**/**tick64Set( )** value, plus all **tickAnnounce( )** calls since.

ERRNO N/A

SEE ALSO **tickLib**, **tickGet( )**, **tick64Set( )**, **tickSet( )**, **tickAnnounce( )**

tick64Set( )

NAME **tick64Set( )** – set the value of the kernel's tick counter in 64 bits

SYNOPSIS

```
void tick64Set
    (
    UINT64 ticks  /* new time in ticks */
    )
```

DESCRIPTION This routine sets the internal tick counter to a specified value in ticks. The new count will be reflected by **tick64Get( )** and **tickGet( )** (only the lower 32 bits), but will not change any delay fields or timeouts selected for any tasks. For example, if a task is delayed for ten ticks, and this routine is called to advance time, the delayed task will still be delayed until ten **tickAnnounce( )** calls have been made.

SMP CONSIDERATIONS

In SMP configuration this API is spinLock restricted meaning that calling this API while holding a spinLock is not allowed. Not comforming to this restriction will potentially lead to a live-lock scenerio.

RETURNS N/A

ERRNO N/A

SEE ALSO **tickLib**, **tick64Get( )**, **tickGet( )**, **tickSet( )**, **tickAnnounce( )**

tickAnnounce( )

NAME **tickAnnounce( )** – announce a clock tick to the kernel

SYNOPSIS

```
void tickAnnounce (void)
```

DESCRIPTION This routine informs the kernel of the passing of time. It should be called from an interrupt service routine that is connected to the system clock. The most common frequencies are 60Hz or 100Hz. Frequencies in excess of 600Hz are an inefficient use of processor power because the system will spend most of its time advancing the clock. By default, this routine is called by **usrClock( )** in **usrConfig.c**.

RETURNS N/A

ERRNO N/A

SEE ALSO **tickLib**, **kernelLib**, **taskLib**, **semLib**, **wdLib**, the VxWorks programmer's guides

tickAnnounceHookAdd( )

NAME **tickAnnounceHookAdd( )** – add a hook routine to be called at each tick interrupt

SYNOPSIS

```
STATUS tickAnnounceHookAdd
    (
    FUNCPTR pFunc
    )
```

DESCRIPTION

This routine adds a hook to perform operations, such as round robin policy implementation, at each tick interrupt. The hooked function must follow the same ISR restrictions and must be callable at interrupt context. The user provided hook routine should be declared as follows:

```
void mySchedulerTickHook
    (
    int         tid             /* interrupt task ID */
    )
```

The user specified hook routines must not access the task structure fields directly. Access routines, such as **taskPriorityGet( )**, should be used to access data structure fields.

RETURNS

OK, or **ERROR** if the hook has been installed

ERRNO

N/A

SEE ALSO

tickLib

tickGet( )

NAME

tickGet( ) – get the value of the kernel's tick counter

SYNOPSIS

```
ULONG tickGet (void)
```

DESCRIPTION

This routine returns the current value of the tick counter. This value is set to zero at startup, incremented by **tickAnnounce( )**, and can be changed using **tickSet( )**.

SMP CONSIDERATIONS

In SMP configuration this API is spinLock restricted meaning that calling this API while holding a spinLock is not allowed. Not comforming to this restriction will potentially lead to a live-lock scenerio.

RETURNS

The most recent **tickSet( )** value, plus all **tickAnnounce( )** calls since.

ERRNO

N/A

SEE ALSO

tickLib, **tickSet( )**, **tickAnnounce( )**

tickSet( )

NAME **tickSet( )** – set the value of the kernel's tick counter

SYNOPSIS

```
void tickSet
    (
    ULONG ticks  /* new time in ticks */
    )
```

DESCRIPTION This routine sets the internal tick counter to a specified value in ticks. The new count will be reflected by **tickGet( )**, but will not change any delay fields or timeouts selected for any tasks. For example, if a task is delayed for ten ticks, and this routine is called to advance time, the delayed task will still be delayed until ten **tickAnnounce( )** calls have been made.

SMP CONSIDERATIONS
In SMP configuration this API is spinLock restricted meaning that calling this API while holding a spinLock is not allowed. Not comforming to this restriction will potentially lead to a live-lock scenerio.

RETURNS N/A

ERRNO N/A

SEE ALSO **tickLib**, **tickGet( )**, **tickAnnounce( )**

timerOpenInit( )

NAME **timerOpenInit( )** – initialize the timer open facility

SYNOPSIS

```
void timerOpenInit (void)
```

DESCRIPTION This routine links the timer creation routine with the open facility into the VxWorks system. It is called automatically when the timer facility is configured into VxWorks by either defining **INCLUDE_OBJ_OPEN** **INCLUDE_POSIX_TIMERS** in **config.h** or selecting **INCLUDE_OBJ_OPEN** and **INCLUDE_POSIX_TIMERS** in the project facility.

RETURNS N/A

ERRNO N/A

SEE ALSO **timerOpen**

timerShowInit( )

NAME **timerShowInit( )** – initialize the timer show routine facility

SYNOPSIS

```
void timerShowInit (void)
```

DESCRIPTION This routine links the timer show routines into the VxWorks system. It is called automatically when the timer show facility is configured into VxWorks using either of the following methods:

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

RETURNS N/A

ERRNO N/A

SEE ALSO **timerShow**

timer_cancel( )

NAME **timer_cancel( )** – cancel a timer

SYNOPSIS

```
int timer_cancel
    (
    timer_t timerid  /* timer ID */
    )
```

DESCRIPTION This routine is a shorthand method of invoking **timer_settime( )**, which stops a timer.

NOTE This is a non-POSIX API.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO **EINVAL**

SEE ALSO **timerLib**

timer_close( )

NAME

timer_close( ) – close a named timer

SYNOPSIS

```
STATUS timer_close
    (
    timer_t timerId  /* timer ID to close */
    )
```

DESCRIPTION

This routine closes a named timer and decrements its reference counter. In case where the counter becomes zero, the timer is deleted if:

- It has been already removed from the name space by a call to **timer_unlink( )**.
- It was created with the **OM_DESTROY_ON_LAST_CALL** option.

NOTE

This is a non-POSIX API. This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_ID_ERROR
The timer ID is invalid.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The timer is not named.

S_objLib_OBJ_DESTROY_ERROR
An error was detected while deleting the timer.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO

timerOpen, **timer_open( )**, **timer_unlink( )**

timer_connect( )

NAME

timer_connect( ) – connect a user routine to the timer signal

SYNOPSIS

```
int timer_connect
```

```
    (
    timer_t     timerid,  /* timer ID      */
    VOIDFUNCPTR routine,  /* user routine  */
    int         arg       /* user argument */
    )
```

DESCRIPTION

This routine sets the specified *routine* to be invoked with *arg* when fielding a signal indicated by the timer's *evp* signal number, or if *evp* is **NULL**, when fielding the default signal (SIGALRM).

The signal handling routine should be declared as:

```
    void my_handler
        (
        timer_t timerid,      /* expired timer ID */
        int     arg           /* user argument    */
        )
```

NOTE

This is a non-POSIX API.

RETURNS

0 (**OK**), or -1 (**ERROR**) if the timer is invalid or cannot bind the signal handler.

ERRNO

EINVAL

SEE ALSO

timerLib

timer_create( )

NAME

timer_create( ) – allocate a timer using the specified clock for a timing base (POSIX)

SYNOPSIS

```
int timer_create
    (
    clockid_t          clock_id,  /* clock ID */
    struct sigevent * evp,        /* user event handler */
    timer_t *          pTimer     /* ptr to return value */
    )
```

DESCRIPTION

This routine returns a value in *pTimer* that identifies the timer in subsequent timer requests. The *evp* argument, if non-**NULL**, points to a **sigevent** structure, which is allocated by the application and defines the signal number and application-specific data to be sent to the task when the timer expires. If *evp* is **NULL**, a default signal (SIGALRM) is queued to the task, and the signal data is set to the timer ID. Initially, the timer is disarmed.

NOTE

If the task that created the timer goes away before the timer expires, the timer expiration process will display a warning message. However, if **INCLUDE_OBJ_OWNERSHIP** is configured, the timer will be deleted at the time the task is deleted and the message will not be displayed.

RETURNS 0 (**OK**), or -1 (**ERROR**) if too many timers already are allocated or the signal number is invalid.

ERRNO

EINVAL

ENOSYS

EAGAIN

S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO **timerLib**, **timer_delete( )**

timer_delete( )

NAME **timer_delete( )** – remove a previously created timer (POSIX)

SYNOPSIS

```
STATUS timer_delete
    (
    timer_t timerid  /* timer ID */
    )
```

DESCRIPTION This routine removes a timer.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO **EINVAL**

SEE ALSO **timerLib**, **timer_create( )**

timer_getoverrun( )

NAME **timer_getoverrun( )** – return the timer expiration overrun (POSIX)

SYNOPSIS

```
int timer_getoverrun
    (
    timer_t timerid  /* timer ID */
    )
```

DESCRIPTION This routine returns the timer expiration overrun count for *timerid*, when called from a timer expiration signal catcher. The overrun count is the number of extra timer expirations that have occurred, up to the implementation-defined maximum **DELAYTIMER_MAX**. If the count is greater than the maximum, it returns the maximum.

RETURNS The number of overruns, or **DELAYTIMER_MAX** if the count equals or is greater than **DELAYTIMER_MAX**, or -1 (**ERROR**) if *timerid* is invalid.

ERRNO **EINVAL**

ENOSYS

SEE ALSO **timerLib**

timer_gettime( )

NAME **timer_gettime( )** – get the remaining time before expiration and the reload value (POSIX)

SYNOPSIS

```
int timer_gettime
    (
    timer_t             timerid,  /* timer ID                        */
    struct itimerspec * value     /* where to return remaining time */
    )
```

DESCRIPTION This routine gets the remaining time and reload value of a specified timer. Both values are copied to the *value* structure.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO **EINVAL**

SEE ALSO **timerLib**

timer_modify( )

NAME **timer_modify( )** – modify a timer

SYNOPSIS

```
STATUS timer_modify
    (
    timer_t           timerId,  /* timer ID */
    struct sigevent * pSigev    /* sigevent describing the notification */
    )
```

DESCRIPTION This routine updates the timer *timerId* with the new notification mechanism as indicated by *pSigev*. This routine should be called in the context of RTP task only, that is, as a system call. This routine should be called with the timer disarmed.

NOTE This is a non-POSIX API.

RETURNS **ERROR** if the *timerId* is invalid or armed, or if there is an error in the notification mechanisim. Otherwise **OK**.

ERRNO **EINVAL**

SEE ALSO **timerLib**

timer_open( )

NAME **timer_open( )** – open a timer

SYNOPSIS

```
timer_t timer_open
    (
    const char *      name,     /* name of timer */
    int               mode,     /* OM_CREATE, ... */
    clockid_t         clockId,  /* clock ID */
    struct sigevent * evp,      /* user event handler */
    void *            context   /* context value */
    )
```

DESCRIPTION This routine opens a timer, which means that it will search the name space and will return the timer_id of an existent timer with same name as *name*, and if none is found, then creates a new one with that name depending on the flags set in the mode parameter. Note that there are two name spaces available to the calling routine in which **timer_open( )** can perform the search, and which are selected depending on the first character in the *name* parameter. When this character is a forward slash **/**, the **public** name space is searched; otherwise the **private** name space is searched. Similarly, if a timer is created, the first character in *name* specifies the name space that contains the timer.

The argument *name* is mandatory. **NULL** or empty strings are not allowed.

Timers created by this routine can not be deleted with **timer_delete( )**. Instead, a **timer_close( )** must be issued for every **timer_open( )**. Then the timer is deleted when it is removed from the name space by a call to **timer_unlink( )**. Alternatively, the timer can be previously removed from the name space, and deleted during the last **timer_close( )**.

A description of the *mode* and *context* arguments follows. See the reference entry for **timer_create( )** for a description of the remaining arguments.

mode
This parameter specifies the timer permissions (not implemented) along with various object management attribute bits as follows:

OM_CREATE
Create a new timer if a matching timer name is not found.

OM_EXCL
When set jointly with **OM_CREATE**, create a new timer immediately without attempting to open an existing timer. An error condition is returned if a timer with *name* already exists. This attribute has no effect if the **OM_CREATE** attribute is not specified.

OM_DELETE_ON_LAST_CLOSE
Only used when a timer is created. If set, the timer will be deleted during the last **timer_close( )** call, independently on whether **timer_unlink( )** was previously called or not.

context
Context value assigned to the created timer. This value is not actually used by VxWorks. Instead, the context value can be used by OS extensions to implement object permissions, for example.

The *clockId* and *evp* are used only when creating a new timer. The clock used by the timer *clockId* is the one defined in *time.h*. The *evp* argument, if non-**NULL**, points to a **sigevent** structure, which is allocated by the application and defines the signal number and application-specific data to be sent to the task when the timer expires. If *evp* is **NULL**, a default signal (SIGALRM) is queued to the task, and the signal data is set to the timer ID. Initially, the timer is disarmed.

NOTE This is a non-POSIX API. This routine is not ISR callable.

SMP CONSIDERATIONS
This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS The timer ID on success; otherwise **ERROR**.

ERRNO **EINVAL**

EAGAIN

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO **timerOpen**, **timer_close( )**, **timer_unlink( )**

timer_settime( )

NAME

timer_settime( ) – set the time until the next expiration and arm timer (POSIX)

SYNOPSIS

```
int timer_settime
    (
    timer_t                     timerid,  /* timer ID                         */
    int                         flags,    /* absolute or relative             */
    const struct itimerspec * value,    /* time to be set                   */
    struct itimerspec       * ovalue    /* prev time set (NULL=no result) */
    )
```

DESCRIPTION

This routine sets the next expiration of the timer, using the **.it_value** of *value*, thus arming the timer. If the timer is already armed, this call resets the time until the next expiration. If **.it_value** is zero, the timer is disarmed.

If *flags* is not equal to **TIMER_ABSTIME**, the interval is relative to the current time, the interval being the **.it_value** of the *value* parameter. If *flags* is equal to **TIMER_ABSTIME**, the expiration is set to the difference between the absolute time of **.it_value** and the current value of the clock associated with *timerid*. If the time has already passed, then the timer expiration notification is made immediately with the signal delivered to the task that created the timer.

The reload value of the timer is set to the value specified by the **.it_interval** field of *value*. When a timer is armed with a nonzero **.it_interval** a periodic timer is set up.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer are rounded up to the larger multiple of the resolution.

If *ovalue* is non-**NULL**, the routine stores a value representing the previous amount of time before the timer would have expired. Or if the timer is disarmed, the routine stores zero, together with the previous timer reload value. The *ovalue* parameter is the same value as that returned by **timer_gettime( )** and is subject to the timer resolution.

WARNING

If **clock_settime( )** is called to reset the absolute clock time after a timer has been set with **timer_settime( )**, and if *flags* is equal to **TIMER_ABSTIME**, then the timer will behave unpredictably. If you must reset the absolute clock time after setting a timer, do not use *flags* equal to **TIMER_ABSTIME**.

RETURNS

0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid, the number of nanoseconds specified by *value* is less than 0 or greater than or equal to 1,000,000,000, or the time specified by *value* exceeds the maximum allowed by the timer.

ERRNO

EINVAL

SEE ALSO

timerLib

timer_show( )

NAME

timer_show( ) – show information on a specified timer

SYNOPSIS

```
int timer_show
    (
    timer_t timerid,  /* timer ID */
    int     verbose   /* Verbose mode: 0, 1 */
    )
```

DESCRIPTION

This routine shows information about the timer specified in *timerid*. If *timerid* is 0 then a list of all timers will be printed. Verbose mode will show additional information about *timerid* including the owner's task name, the timers type, and the state of the timer.

EXAMPLE

->timer_show (0,0)

timerid taskId evp routine arg Remaining Period ---------- ---------- ---------- ---------- ----- ---------- ---------- 0x6170bf20 0x6170bba0 0x6170bf74 0x604ce330 1617182668 0.000000 0.000000 0x603b56b8 0x6170bba0 0x603b570c 0x604ce330 1617182668 0.000000 0.000000 0x617115e0 0x6170bba0 0x61711634 0x604ce330 1617182668 0.000000 0.000000

->timer_show (0x6170bf20,0)

timerid taskId evp routine arg Remaining Period ---------- ---------- ---------- ---------- ----- ---------- ---------- 0x6170bf20 0x6170bba0 0x6170bf74 0x604ce330 1617182668 0.000000 0.000000

->timer_show (0x6170bf20,1)

timerid taskId evp routine arg Remaining Period ---------- ---------- ---------- ---------- ----- ---------- ---------- 0x6170bf20 0x6170bba0 0x6170bf74 0x604ce330 1617182668 0.000000 0.000000 Owners Task Name: tTimer_gettimeTest3 Type of Timer: **CLOCK_REALTIME** State: Active

WARNING

This is a non-POSIX API.

RETURNS

0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid, or the context is invalid.

ERRNO

N/A

SEE ALSO

timerShow

timer_unlink( )

NAME

timer_unlink( ) – unlink a named timer

SYNOPSIS

```
STATUS timer_unlink
    (
    const char * name  /* name of the timer to unlink */
    )
```

DESCRIPTION

This routine removes a timer from the name space, and marks it as ready for deletion on the last **timer_close( )**. In case there are already no outstanding **timer_open( )** calls, the timer is deleted. After a timer is unlinked, subsequent calls to **timer_open( )** using *name* will not be able to find the timer, even if it has not been deleted yet. Instead, a new timer could be created if **timer_open( )** is called with the **OM_CREATE** flag.

NOTE

This is a non-POSIX API. This routine is not ISR callable.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted. These restrictions are not enforced by the implementation so it is the responsibility of the caller to ensure they are complied with. Future implementations may enforce these restrictions.

RETURNS

OK, or **ERROR** if unsuccessful

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL** or empty.

S_objLib_OBJ_NOT_FOUND
No timer with *name* was found.

S_objLib_OBJ_DESTROY_ERROR
Error while deleting the timer.

S_intLib_NOT_ISR_CALLABLE
This routine must not be called from an ISR.

SEE ALSO

timerOpen, **timer_open( )**, **timer_close( )**

timex( )

NAME

timex( ) – time a single execution of a function or functions

SYNOPSIS

```
void timex
```

```
(
FUNCPTR func,  /* function to time (optional)                 */
int     arg1,  /* first of up to 8 args to call function with (opt) */
int     arg2,
int     arg3,
int     arg4,
int     arg5,
int     arg6,
int     arg7,
int     arg8
)
```

DESCRIPTION This routine times a single execution of a specified function with up to eight of the function's arguments. If no function is specified, it times the execution of the current list of functions to be timed, which is created using **timexFunc( )**, **timexPre( )**, and **timexPost( )**. If **timex( )** is executed with a function argument, the entire current list is replaced with the single specified function.

When execution is complete, **timex( )** displays the execution time. If the execution was so fast relative to the clock rate that the time is meaningless (error > 50%), a warning message is printed instead. In such cases, use **timexN( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**, **timexFunc( )**, **timexPre( )**, **timexPost( )**, **timexN( )**

timexClear( )

NAME **timexClear( )** – clear the list of function calls to be timed

SYNOPSIS

```
void timexClear (void)
```

DESCRIPTION This routine clears the current list of functions to be timed.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**

timexFunc( )

NAME **timexFunc( )** – specify functions to be timed

SYNOPSIS

```
void timexFunc
    (
    int     i,      /* function number in list (0..3)               */
    FUNCPTR func,   /* function to be added (NULL if to be deleted) */
    int     arg1,   /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine adds or deletes functions in the list of functions to be timed as a group by calls to **timex( )** or **timexN( )**. Up to four functions can be included in the list. The argument *i* specifies the function's position in the sequence of execution (0, 1, 2, or 3). A function is deleted by specifying its sequence number *i* and **NULL** for the function argument *func*.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**, **timex( )**, **timexN( )**

timexHelp( )

NAME **timexHelp( )** – display synopsis of execution timer facilities

SYNOPSIS

```
void timexHelp (void)
```

DESCRIPTION This routine displays the following summary of the available execution timer functions:

```
timexHelp                          Print this list.
timex       [func,[args...]]   Time a single execution.
timexN      [func,[args...]]   Time repeated executions.
timexClear                         Clear all functions.
timexFunc   i,func,[args...]   Add timed function number i (0,1,2,3).
timexPre    i,func,[args...]   Add pre-timing function number i.
timexPost   i,func,[args...]   Add post-timing function number i.
timexShow                          Show all functions to be called.
```

```
Notes:
  1) timexN() will repeat calls enough times to get
     timing accuracy to approximately 2%.
  2) A single function can be specified with timex() and timexN();
     or, multiple functions can be pre-set with timexFunc().
  3) Up to 4 functions can be pre-set with timexFunc(),
     timexPre(), and timexPost(), i.e., i in the range 0 - 3.
  4) timexPre() and timexPost() allow locking/unlocking, or
     raising/lowering priority before/after timing.
```

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**

timexInit( )

NAME **timexInit( )** – include the execution timer library

SYNOPSIS

```
void timexInit (void)
```

DESCRIPTION This null routine is provided so that **timexLib** can be linked into the system. If the configuration macro **INCLUDE_TIMEX** is defined, it is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**

timexN( )

NAME **timexN( )** – time repeated executions of a function or group of functions

SYNOPSIS

```
void timexN
    (
    FUNCPTR func,  /* function to time (optional)                 */
    int     arg1,  /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine times the execution of the current list of functions to be timed in the same manner as **timex( )**; however, the list of functions is called a variable number of times until sufficient resolution is achieved to establish the time with an error less than 2%. (Since each iteration of the list may be measured to a resolution of +/- 1 clock tick, repetitive timings decrease this error to 1/N ticks, where N is the number of repetitions.)

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**, **timexFunc( )**, **timex( )**

timexPost( )

NAME **timexPost( )** – specify functions to be called after timing

SYNOPSIS

```
void timexPost
    (
    int     i,     /* function number in list (0..3)               */
    FUNCPTR func,  /* function to be added (NULL if to be deleted) */
    int     arg1,  /* first of up to 8 args to call function with  */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine adds or deletes functions in the list of functions to be called immediately following the timed functions. A maximum of four functions may be included. Up to eight arguments may be passed to each function.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**

timexPre( )

NAME **timexPre( )** – specify functions to be called prior to timing

SYNOPSIS

```
void timexPre
    (
    int     i,     /* function number in list (0..3)              */
    FUNCPTR func,  /* function to be added (NULL if to be deleted) */
    int     arg1,  /* first of up to 8 args to call function with  */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine adds or deletes functions in the list of functions to be called immediately prior to the timed functions. A maximum of four functions may be included. Up to eight arguments may be passed to each function.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**

timexShow( )

NAME **timexShow( )** – display the list of function calls to be timed

SYNOPSIS

```
void timexShow (void)
```

DESCRIPTION This routine displays the current list of function calls to be timed. These lists are created by calls to **timexPre( )**, **timexFunc( )**, and **timexPost( )**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **timexLib**, **timexPre( )**, **timexFunc( )**, **timexPost( )**

tlsTaskInit( )

NAME **tlsTaskInit( )** – Thread Local Storage init routine

SYNOPSIS

```
STATUS tlsTaskInit (void)
```

DESCRIPTION This routine initializes the current task thread local storage for all available modules. Once this is called, any access of a __thread variable of any of those modules will be deterministic. If this routine is not called, access to the __thread variable of a module may take longer the first time because of time needed to allocate memory to manage the module's __thread variables.

RETURNS **OK**, or **ERROR**

ERRNO N/A

SEE ALSO **tlsLib**

tr( )

NAME **tr( )** – resume a task

SYNOPSIS

```
void tr
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION This command resumes the execution of a suspended task. It simply calls **taskResume( )**.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **ts( )**, **taskResume( )**, the VxWorks programmer guides.

traceTmrResolutionGet( )

NAME **traceTmrResolutionGet( )** – get resolution of timestamp source, in nanoseconds

SYNOPSIS

```
STATUS traceTmrResolutionGet
    (
    struct timespec * pTimestamp  /* destination for resolution */
    )
```

DESCRIPTION Write the resolution of the timestamp source into the supplied timespec. If the timestamp resolution cannot be obtained (maybe the timestamp driver is not available) then the timespec structure will be filled with zeroes.

RETURNS **OK**, or **ERROR**.

ERRNO Not Available

SEE ALSO **wvTmrLib**

transCommit( )

NAME **transCommit( )** – externally-callable function to do a commit

SYNOPSIS

```
int transCommit
    (
    TRANS_XBD * pDev
    )
```

DESCRIPTION The *pDev* argument is the device handle (as provided to the warning and panic hooks).

This function is only for special cases (such as "automatic commits" on a **WARN_OUT_OF_UNITS** warning, which are generally advised against for data-control reasons).

RETURNS error number, or 0 (**OK**) on success.

ERRNO N/A

SEE ALSO **xbdTrans**, **usrTransCommit( )**, **usrTransCommitFd( )**.

transDevCreate( )

NAME

transDevCreate( ) – create a transactional XBD.

SYNOPSIS

```
TRANS_XBD *transDevCreate
    (
    device_t subDev  /* lower level device */
    )
```

DESCRIPTION

THIS ROUTINE IS EXTERNALLY-VISIBLE FOR BACKWARDS COMPATIBILITY ONLY.

This is essentially an internal version that gives you the pointer to the device. The pointer becomes invalid when the device is ejected, so one should use xbdTransDevCreate instead (to get a device_t that can be checked).

RETURNS

TRANS_XBD *, or **NULL** on failure.

ERRNO

Not Available

SEE ALSO

xbdTrans

trgAdd( )

NAME

trgAdd( ) – add a new trigger to the trigger list

SYNOPSIS

```
TRIGGER_ID trgAdd
    (
    event_t   event,
    int       status,
    int       contextType,
    UINT32    contextId,
    OBJ_ID    objId,
    int       conditional,
    int       condType,
    int     * condEx1,
    int       condOp,
    int       condEx2,
    BOOL      disable,
    TRIGGER * chain,
    int       actionType,
    FUNCPTR   actionFunc,
    BOOL      actionDef,
    int       actionArg
    )
```

DESCRIPTION

This routine creates a new trigger and adds it to the proper trigger list. It takes the following parameters:

event
: as defined in **eventP.h** for System Viewer, if given.

status
: the initial status of the trigger (enabled or disabled).

contextType
: the type of context where the event occurs.

contextId
: the ID (if any) of the context where the event occurs.

objectId
: if given and applicable.

conditional
: the indicator that there is a condition on the trigger.

condType
: the indicator that the condition is either a variable or a function.

condEx1
: the first element in the comparison.

condOp
: the type of operator (==, !=, , <=,, >=, |, &).

condEx2
: the second element in the comparison (a constant).

disable
: the indicator of whether the trigger must be disabled once it is hit.

chain
: a pointer to another trigger associated to this one (if any).

actionType
: the type of action associated with the trigger (none, func, lib).

actionFunc
: the action associated with the trigger (the function).

actionDef
: the indicator of whether the action can be deferred (deferred is the default).

actionArg
: the argument passed to the function, if any.

Attempting to call trgAdd whilst triggering is enabled is not allowed and will return **NULL**.

RETURNS **TRIGGER_ID**, or **NULL** if either the trigger ID can not be allocated, or if called whilst triggering is enabled.

ERRNO **S_intLib_NOT_ISR_CALLABLE**
S_objLib_OBJ_ID_ERROR
S_memLib_NOT_ENOUGH_MEMORY
S_memLib_BLOCK_ERROR
S_taskLib_ILLEGAL_PRIORITY

SEE ALSO **trgLib**, **trgDelete( )**

trgChainSet( )

NAME **trgChainSet( )** – chains two triggers

SYNOPSIS

```
STATUS trgChainSet
    (
    TRIGGER_ID fromId,
    TRIGGER_ID toId
    )
```

DESCRIPTION This routine chains two triggers together. When the first trigger fires, it calls **trgEnable( )** for the second trigger. The second trigger must be created disabled in order to maintain the correct sequence.

RETURNS **OK** or **ERROR**.

ERRNO

SEE ALSO **trgLib**, **trgEnable( )**

trgDelete( )

NAME **trgDelete( )** – delete a trigger from the trigger list

SYNOPSIS

```
STATUS trgDelete
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION This routine deletes a trigger by removing it from the trigger list. It also checks that no other triggers are still active. If there are no active triggers and triggering is still on, it turns triggering off.

RETURNS **OK**, or **ERROR** if the trigger is not found.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **trgLib**, **trgAdd( )**

trgDisable( )

NAME **trgDisable( )** – turn a trigger off

SYNOPSIS
```
STATUS trgDisable
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION This routine disables a trigger. It also checks to see if there are triggers still active. If this is the last active trigger it sets triggering off.

RETURNS **OK**, or **ERROR** if the trigger ID is not found.

ERRNO

SEE ALSO **trgLib**, **trgEnable( )**

trgEnable( )

NAME **trgEnable( )** – enable a trigger

SYNOPSIS
```
STATUS trgEnable
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION This routine enables a trigger that has been created with **trgAdd( )**. A counter is incremented to keep track of the total number of enabled triggers so that **trgDisable( )** knows when to set triggering off. If the maximum number of enabled triggers is reached, an error is returned.

RETURNS
OK, or **ERROR** if the trigger ID is not found or if the maximum number of triggers has already been enabled.

ERRNO

SEE ALSO
trgLib, **trgDisable( )**

trgEvent( )

NAME
trgEvent( ) – trigger a user-defined event

SYNOPSIS

```
void trgEvent
    (
    event_t evtId  /* event */
    )
```

DESCRIPTION
This routine triggers a user event. A trigger must exist and triggering must have been started with **trgOn( )** or from the triggering GUI to use this routine. The *evtId* should be in the range 40000-65535.

RETURNS
N/A

ERRNO

SEE ALSO
trgLib, **dbgLib**, **e( )**

trgLibInit( )

NAME
trgLibInit( ) – initialize the triggering library

SYNOPSIS

```
STATUS trgLibInit (void)
```

DESCRIPTION
This routine initializes the trigger class. Triggers are VxWorks objects and therefore require a class to be initialized.

RETURNS
OK or **ERROR**.

ERRNO

SEE ALSO **trgLib**

trgOff( )

NAME **trgOff( )** – set triggering off

SYNOPSIS

```
void trgOff (void)
```

DESCRIPTION This routine turns triggering off. From this time on, when an event point is hit, no search on triggers is performed.

RETURNS N/A

ERRNO

SEE ALSO **trgLib**, **trgOn( )**

trgOn( )

NAME **trgOn( )** – set triggering on

SYNOPSIS

```
STATUS trgOn (void)
```

DESCRIPTION This routine activates triggering. From this time on, any time an event point is hit, a check for the presence of possible triggers is performed. Start triggering only when needed since some overhead is introduced.

NOTE If **trgOn( )** is called when there are no triggers in the trigger list, it immediately sets triggering off again. If **trgOn( )** is called with at least one trigger in the list, triggering begins. Triggers should not be added to the list while triggering is on since this can create instability.

RETURNS **OK** or **ERROR**.

ERRNO

SEE ALSO **trgLib**, **trgOff( )**

trgReset( )

NAME

trgReset( ) – Reset a trigger in the trigger list

SYNOPSIS

```
STATUS trgReset
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION

This routine resets a trigger. It sets the triggers hit count to zero and sets its state back to the initial state saved when the trigger was downloaded

RETURNS

OK, or **ERROR** if the trigger is not found.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

trgLib

trgShow( )

NAME

trgShow( ) – show trigger information

SYNOPSIS

```
STATUS trgShow
    (
    TRIGGER_ID trgId,  /* trigger id to show, or NULL for all triggers */
    int        level   /* detail level: 1 gives more detail */
    )
```

DESCRIPTION

This routine displays trigger information. If *trgId* is passed, only the summary for that trigger is displayed. If no parameter is passed, the list of existing triggers is displayed with a summary of their state. For example:

```
   trgID      Status   EvtID    ActType Action     Dis    Chain
   ------------------------------------------------------------
   0xffedfc   disabled  101           3 0x14e7a4     Y 0xffe088
   0xffe088   enabled    55           1 0x10db58     Y      0x0
```

If *level* is 1, then more detailed information is displayed.

EXAMPLE

```
    -> trgShow trgId, 1
```

RETURNS

OK.

ERRNO

Not Available

SEE ALSO **trgShow**, **trgLib**

trgShowInit( )

NAME **trgShowInit( )** – initialize the trigger show facility

SYNOPSIS

```
void trgShowInit (void)
```

DESCRIPTION This routine links the trigger show facility into the VxWorks system. These routines are included automatically when **INCLUDE_TRIGGER_SHOW** is defined.

RETURNS N/A

ERRNO Not Available

SEE ALSO **trgShow**

trgWorkQReset( )

NAME **trgWorkQReset( )** – Resets the trigger work queue task and queue

SYNOPSIS

```
STATUS trgWorkQReset (void)
```

DESCRIPTION When a trigger fires, if the assocated action requires a function to be called in "safe" mode, a pointer to the required function will be placed on a queue known as the "triggering work queue". A system task "tActDef" is spawned to action these requests at task level. Should the user have need to reset this work queue (e.g. if a called task causes an exception which causes the trgActDef task to be SUSPENDED, or if the queue gets out of sync and becomes unresponsive), **trgWorkQReset( )** may be called.

Its effect is to delete the trigger work queue task and its associated resources and then recreate them. Any entries pending on the triggering work queue will be lost. Calling this function with triggering on will result in triggering being turned off before the queue reset takes place. It is the responsibility of the user to turn triggering back on.

This function may not be called from interrupt.

RETURNS **OK**, or **ERROR** if the triggering task and its associated resources cannot be deleted and recreated.

ERRNO **S_taskLib_NAME_NOT_FOUND**
S_taskLib_ILLEGAL_PRIORITY
S_intLib_NOT_ISR_CALLABLE
S_objLib_OBJ_ID_ERROR
S_memLib_NOT_ENOUGH_MEMORY
S_memLib_BLOCK_ERROR

SEE ALSO **trgLib**

trunc( )

NAME **trunc( )** – truncate to integer

SYNOPSIS

```
double trunc
    (
    double x  /* value to truncate */
    )
```

DESCRIPTION This routine discards the fractional part of a double-precision value *x*.

RETURNS The integer portion of *x*, represented in double-precision.

ERRNO Not Available

SEE ALSO **mathALib**

truncf( )

NAME **truncf( )** – truncate to integer

SYNOPSIS

```
float truncf
    (
    float x  /* value to truncate */
    )
```

DESCRIPTION This routine discards the fractional part of a single-precision value *x*.

RETURNS The integer portion of *x*, represented in single precision.

ERRNO Not Available

SEE ALSO **mathALib**

ts( )

NAME **ts( )** – suspend a task

SYNOPSIS

```
void ts
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION This command suspends the execution of a specified task. It simply calls **taskSuspend( )**.

RETURNS N/A

ERRNO N/A

SEE ALSO **usrLib**, **tr( )**, **taskSuspend( )**, the VxWorks programmer guides.

tsecRegister( )

NAME **tsecRegister( )** – register with the VxBus subsystem

SYNOPSIS

```
void tsecRegister(void)
```

DESCRIPTION This routine registers the TSEC driver with VxBus as a child of the PLB bus type.

RETURNS N/A

ERRNO N/A

SEE ALSO **tsecVxbEnd**

tt( )

NAME

tt( ) – display a stack trace of a task

SYNOPSIS

```
STATUS tt
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION

This routine displays a list of the nested routine calls that the specified task is in. Each routine call and its parameters are shown.

If *taskNameOrId* is not specified or zero, the last task referenced is assumed. The **tt( )** routine can only trace the stack of a task other than itself and unbreakable tasks. For instance, when **tt( )** is called from the shell, it cannot trace the shell's stack.

EXAMPLE

```
   -> tt "tAioIoTask1"
    0x600f5cb6 aioIoTask +0x16: 0x60129c34 ([0x60459b08, 0xffffffff, 0, 0,
0])
    0x60129d33 semTake   +0x123: 0x601291d3 (0x60459b08, 0xffffffff)
   value = 0 = 0x0
```

This indicates that **tAioIoTask1( )** is currently in **semTake( )** (with two parameters) and was called by **aioIoTask( )** (with five parameters).

CAVEAT

In order to do the trace, some assumptions are made. In general, the trace will work for all C language routines and for assembly language routines. Depending of the architecture and at which point the task is suspended, the trace facility may produce inaccurate results or fail completely. Moreover, if the routine is written in a language other than C, the routine's entry point is non-standard, or the task's stack is corrupted, the trace facility may produce inaccurate results too. Also, all parameters are assumed to be 32-bit quantities, so structures passed as parameters will be displayed as *long* integers.

RETURNS

OK, or **ERROR** if the task does not exist.

ERRNO

N/A

SEE ALSO

dbgLib, the VxWorks programmer guides, *VxWorks Command-Line Tools User's Guide*.

ttyDevCreate( )

NAME

ttyDevCreate( ) – create a VxWorks device for a serial channel

SYNOPSIS

```
STATUS ttyDevCreate
    (
    char *     name,       /* name to use for this device      */
    SIO_CHAN * pSioChan,   /* pointer to core driver structure */
    int        rdBufSize,  /* read buffer size, in bytes       */
    int        wrtBufSize  /* write buffer size, in bytes      */
    )
```

DESCRIPTION

This routine creates a device on a specified serial channel. Each channel to be used should have exactly one device associated with it by calling this routine.

For instance, to create the device "/tyCo/0", with buffer sizes of 512 bytes, the proper call would be:

```
    ttyDevCreate ("/tyCo/0", pSioChan, 512, 512);
```

Where *pSioChan* is the address of the underlying **SIO_CHAN** serial channel descriptor (defined in **sioLib.h**). This routine is called automatically when **INCLUDE_TTY_DEV** is configured in VxWorks. It initializes two channels using the default names /tyCo/0 and /tyCo/1.

RETURNS

OK, or **ERROR** if the driver is not installed, or the device already exists.

ERRNO

S_ioLib_NO_DRIVER (**ENXIO**)
The **ttyDrv** driver is not installed in system.

S_iosLib_DUPLICATE_DEVICE_NAME (**EINVAL**)
Device name already in use.

SEE ALSO

ttyDrv

ttyDrv( )

NAME

ttyDrv( ) – initialize the *tty* driver

SYNOPSIS

```
STATUS ttyDrv (void)
```

DESCRIPTION

This routine initializes the *tty* driver, which is the OS interface to core serial channel(s). It is called automatically when **INCLUDE_TTY_DEV** is configured in VxWorks.

After this routine is called, **ttyDevCreate( )** is typically called to bind serial channels to VxWorks devices.

RETURNS **OK**, or **ERROR** if the driver cannot be installed.

ERRNO N/A

SEE ALSO **ttyDrv**

tw( )

NAME **tw( )** – print info about the object the given task is pending on

SYNOPSIS

```
void tw
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION This routine shows task's pending information of the task *taskNameOrId*.

This routine doesn't support POSIX semaphores and message queues. This command doesn't support pending signals.

List of object types that are recognized:

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrLib**, **w( )**, the VxWorks programmer guides.

tyAbortFuncSet( )

NAME **tyAbortFuncSet( )** – set the abort function

SYNOPSIS

```
void tyAbortFuncSet
    (
    FUNCPTR func  /* routine to call when abort char received */
    )
```

DESCRIPTION

This routine sets the function that will be called when the abort character is received on a *tty*. There is only one global abort function, used for any *tty* on which **OPT_ABORT** is enabled. When the abort character is received from a *tty* with **OPT_ABORT** set, the function specified in *func* will be called, with no parameters, from interrupt level.

Setting an abort function of **NULL** will disable the abort function.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

tyLib, **tyAbortSet( )**

tyAbortGet( )

NAME

tyAbortGet( ) – get the abort character

SYNOPSIS

```
char tyAbortGet (void)
```

DESCRIPTION

This routine returns the abort character.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

tyLib, **tyAbortFuncSet( )**, **tyAbortSet( )**

tyAbortSet( )

NAME

tyAbortSet( ) – change the abort character

SYNOPSIS

```
void tyAbortSet
    (
    char ch  /* char to be abort */
    )
```

DESCRIPTION

This routine sets the abort character to *ch*. The default abort character is CTRL-C.

Typing the abort character to any device whose **OPT_ABORT** option is set will cause the shell task to be killed and restarted. Note that the character set by this routine applies to all devices whose handlers use the standard *tty* package **tyLib**.

RETURNS N/A

ERRNO N/A

SEE ALSO **tyLib**, **tyAbortFuncSet( )**, **tyAbortGet( )**

tyBackspaceSet( )

NAME **tyBackspaceSet( )** – change the backspace character

SYNOPSIS

```
void tyBackspaceSet
    (
    char ch  /* char to be backspace */
    )
```

DESCRIPTION This routine sets the backspace character to *ch*. The default backspace character is CTRL-H.

Typing the backspace character to any device operating in line protocol mode (**OPT_LINE** set) will cause the previous character typed to be deleted, up to the beginning of the current line. Note that the character set by this routine applies to all devices whose handlers use the standard *tty* package **tyLib**.

RETURNS N/A

ERRNO N/A

SEE ALSO **tyLib**

tyDeleteLineSet( )

NAME **tyDeleteLineSet( )** – change the line-delete character

SYNOPSIS

```
void tyDeleteLineSet
    (
    char ch  /* char to be line-delete */
    )
```

DESCRIPTION

This routine sets the line-delete character to *ch*. The default line-delete character is CTRL-U.

Typing the delete character to any device operating in line protocol mode (**OPT_LINE** set) will cause all characters in the current line to be deleted. Note that the character set by this routine applies to all devices whose handlers use the standard *tty* package **tyLib**.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

tyLib

tyDevInit( )

NAME

tyDevInit( ) – initialize the *tty* device descriptor

SYNOPSIS

```
STATUS tyDevInit
    (
    FAST TY_DEV_ID pTyDev,      /* ptr to tty dev descriptor to init */
    int            rdBufSize,   /* size of read buffer in bytes     */
    int            wrtBufSize,  /* size of write buffer in bytes    */
    FUNCPTR        txStartup    /* device transmit start-up routine */
    )
```

DESCRIPTION

This routine initializes a *tty* device descriptor according to the specified parameters. The initialization includes allocating read and write buffers of the specified sizes from the memory pool, and initializing their respective buffer descriptors. The semaphores are initialized and the write semaphore is given to enable writers. Also, the transmitter start-up routine pointer is set to the specified routine. All other fields in the descriptor are zeroed.

This routine should be called only by serial drivers.

RETURNS

OK, or **ERROR** if there is not enough memory to allocate data structures.

ERRNO

N/A

SEE ALSO

tyLib

tyDevRemove( )

NAME

tyDevRemove( ) – remove the *tty* device descriptor

SYNOPSIS

```
STATUS tyDevRemove
    (
    TY_DEV_ID pTyDev  /* ptr to tty dev descriptor to remove */
    )
```

DESCRIPTION

This routine removes an existing *tty* device descriptor. It releases the read and write buffers and the descriptor data structure.

RETURNS

OK, or **ERROR** if expected data structures are not found

ERRNO

N/A.

SEE ALSO

tyLib

tyDevTerminate( )

NAME

tyDevTerminate( ) – terminate the *tty* device descriptor

SYNOPSIS

```
STATUS tyDevTerminate
    (
    TY_DEV_ID pTyDev  /* ptr to tty dev descriptor to terminate */
    )
```

DESCRIPTION

This routine terminates a *tty* device descriptor.   The termination includes freeing memory for the read and write buffers, and terminating the various semaphores.

This routine should be called only by serial drivers.

RETURNS

OK

ERRNO

Not Available

SEE ALSO

tyLib

tyEOFGet( )

NAME

tyEOFGet( ) – get the current end-of-file character

SYNOPSIS

```
char tyEOFGet (void)
```

DESCRIPTION

This routine returns the current end-of-file character.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

tyLib, **tyEOFSet( )**

tyEOFSet( )

NAME

tyEOFSet( ) – change the end-of-file character

SYNOPSIS

```
void tyEOFSet
    (
    char ch  /* char to be EOF */
    )
```

DESCRIPTION

This routine sets the **EOF** character to *ch*. The default **EOF** character is CTRL-D.

Typing the **EOF** character to any device operating in line protocol mode (**OPT_LINE** set) will cause no character to be entered in the current line, but will cause the current line to be terminated (thus without a newline character). The line is made available to reading tasks. Thus, if the **EOF** character is the first character input on a line, a line length of zero characters is returned to the reader. This is the standard end-of-file indication on a read call. Note that the **EOF** character set by this routine will apply to all devices whose handlers use the standard *tty* package **tyLib**.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

tyLib

tyIRd( )

NAME

tyIRd( ) – interrupt-level input

SYNOPSIS

```
STATUS tyIRd
    (
    FAST TY_DEV_ID pTyDev,  /* ptr to tty device descriptor */
    FAST char      inchar   /* character read */
    )
```

DESCRIPTION

This routine handles interrupt-level character input for *tty* devices. A device driver calls this routine when it has received a character. This routine adds the character to the ring buffer for the specified device, and gives a semaphore if a task is waiting for it.

This routine also handles all the special characters, as specified in the option word for the device, such as X-on, X-off, NEWLINE, or backspace.

RETURNS

OK, or **ERROR** if the ring buffer is full.

ERRNO

N/A

SEE ALSO

tyLib

tyITx( )

NAME

tyITx( ) – interrupt-level output

SYNOPSIS

```
STATUS tyITx
    (
    FAST TY_DEV_ID pTyDev,  /* pointer to tty device descriptor */
    char           *pChar   /* where to put character to be output */
    )
```

DESCRIPTION

This routine gets a single character to be output to a device. It looks at the ring buffer for *pTyDev* and gives the caller the next available character, if there is one. The character to be output is copied to *pChar*.

RETURNS

OK if there are more characters to send, or **ERROR** if there are no more characters.

ERRNO

N/A

SEE ALSO

tyLib

tyIoctl( )

NAME **tyIoctl( )** – handle device control requests

SYNOPSIS

```
STATUS tyIoctl
    (
    FAST TY_DEV_ID pTyDev,   /* ptr to device to control */
    int            request,  /* request code             */
    int            arg       /* some argument            */
    )
```

DESCRIPTION This routine handles **ioctl( )** requests for *tty* devices. The I/O control functions for *tty* devices are described in the reference entry for **tyLib**.

BUGS In line protocol mode (**OPT_LINE** option set), the FIONREAD function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines consisting of just NEWLINEs were in the input buffer, the FIONREAD function would return the value ten (five characters + five lines).

RETURNS **OK** or **ERROR**.

ERRNO N/A

SEE ALSO **tyLib**

tyLibInit( )

NAME **tyLibInit( )** – initialize the *tty* library

SYNOPSIS

```
STATUS tyLibInit
    (
    int xoffPercent,  /* default Buffer percentage for sending XOFF */
    int xonPercent,   /* default Buffer percentage for sending XON */
    int wrtThreshold  /* default Buffer count for enabling other senders */
    )
```

DESCRIPTION This routine initialized the library, and set the threshold values that will be used for xoff/xon control and for enabling new writer tasks.

The xoff/xon threshold values are specified as percentages of buffer fill when the related action will occur. Normally xoff is sent out the transmit port when the receive buffer is 85% full. The xon character will be sent to the transmit port when the buffer has drained down to 50% the normal default value for the xonThreshold.

The wrtThreshold is specified in the number of free bytes that must exist in the transmit buffer before a new writer task will be awakened. If the transmit buffer is very full, enabling a new transmit task can potentially awaken all waiting tasks and cause them all to priority arbitrate for the device only to find it blocked again. This threshold prevents the waking up of all pended tasks unless there is some minimum amount of space left in the transmit buffer. Pending tasks will be awakened when the buffer has drained to a point below the threshold number of characters.

RETURNS **OK** or **ERROR** if arguments are invalid. The xon percentage must be smaller than the xoff percentage. Both must be in the range of 1 to 99.

ERRNO Not Available

SEE ALSO **tyLib**

tyMonitorTrapSet( )

NAME **tyMonitorTrapSet( )** – change the trap-to-monitor character

SYNOPSIS

```
void tyMonitorTrapSet
    (
    char ch  /* char to be monitor trap */
    )
```

DESCRIPTION This routine sets the trap-to-monitor character to *ch*. The default trap-to-monitor character is CTRL-X.

Typing the trap-to-monitor character to any device whose **OPT_MON_TRAP** option is set will cause the resident ROM monitor to be entered, if one is present. Once the ROM monitor is entered, the normal multitasking system is halted.

Note that the trap-to-monitor character set by this routine will apply to all devices whose handlers use the standard *tty* package **tyLib**. Also note that not all systems have a monitor trap available.

RETURNS N/A

ERRNO N/A

SEE ALSO **tyLib**

tyRead( )

NAME

tyRead( ) – do a task-level read for a *tty* device

SYNOPSIS

```
int tyRead
    (
    FAST TY_DEV_ID pTyDev,   /* device to read          */
    char           *buffer,  /* buffer to read into     */
    int            maxbytes  /* maximum length of read */
    )
```

DESCRIPTION

This routine handles the task-level portion of the *tty* handler's read function. It reads into the buffer up to *maxbytes* available bytes.

This routine should only be called from serial device drivers.

RETURNS

The number of bytes actually read into the buffer.

ERRNO

N/A

SEE ALSO

tyLib

tyWrite( )

NAME

tyWrite( ) – do a task-level write for a *tty* device

SYNOPSIS

```
int tyWrite
    (
    FAST TY_DEV_ID pTyDev,   /* ptr to device structure */
    char           *buffer,  /* buffer of data to write  */
    FAST int       nbytes    /* number of bytes in buffer */
    )
```

DESCRIPTION

This routine handles the task-level portion of the *tty* handler's write function.

RETURNS

The number of bytes actually written to the device.

ERRNO

N/A

SEE ALSO

tyLib

tyXoffHookSet( )

NAME

tyXoffHookSet( ) – install a hardware flow control function

SYNOPSIS

```
STATUS tyXoffHookSet
    (
    TY_DEV_ID pTyDev,  /* pointer to device structure */
    FUNCPTR   func,    /* Hardware flow control routine */
    int       arg      /* First argument to func routine */
    )
```

DESCRIPTION

This routine installs a hook routine to implement incoming flow control, to replace the default software XOFF/XON flow control method.

The installed function will be called with two arguments. The first is the arg provided when the hook is set, and the second is a boolean value to indicate that incoming characters should be stopped (**TRUE** means to disable input, **FALSE** means to allow input).

```
    VOID func (int arg, BOOLEAN xoffValue);
```

Installing a **NULL** function pointer will restore the default software XOFF/XON method. This will nullify any previous XoffHook installation.

With any change to the flow control routine, the old flow control routine is invoked to enable flow, before the new routine is actually installed. This insures that the incoming flow is not locked up when the method is changed.

RETURNS

OK if successful, or **ERROR** if the **TY_DEV_ID** is invalid.

ERRNO

Not Available

SEE ALSO

tyLib

unixDiskDevCreate( )

NAME

unixDiskDevCreate( ) – create a UNIX disk device

SYNOPSIS

```
BLK_DEV * unixDiskDevCreate
    (
    char * unixFile,      /* name of the UNIX file                 */
    int    bytesPerBlk,   /* number of bytes per block             */
    int    blksPerTrack,  /* number of blocks per track            */
    int    nBlocks        /* number of blocks on this device       */
    )
```

DESCRIPTION

This routine creates a UNIX disk device.

The *unixFile* parameter specifies the name of the UNIX file to use for the disk device.

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 disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (i.e., the disk is defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, a default size is used. The default is calculated as the size of the UNIX disk divided by the number of bytes per block.

This routine is only applicable to VxSim for Solaris.

IMPORTANT NOTE

This routine is obsolete, but is kept for backward compatibility with previous version.

RETURNS

A pointer to block device (**BLK_DEV**) structure, or **NULL**, if unable to open the UNIX disk.

ERRNO

Not Available

SEE ALSO

unixDrv

unixDiskInit( )

NAME

unixDiskInit( ) – initialize a dosFs disk on top of UNIX

SYNOPSIS

```
void unixDiskInit
    (
    char * unixFile,  /* UNIX file name */
    char * volName,   /* dosFs name */
    int    diskSize   /* number of bytes */
    )
```

DESCRIPTION

This routine provides some convenience for a user wanting to create a UNIX disk-based dosFs file system under VxWorks. The user only specifes the UNIX file to use, the dosFs volume name, and the size of the volume in bytes, if the UNIX file needs to be created.

This routine is only applicable to VxSim for Solaris.

IMPORTANT NOTE

This routine is obsolete, but is kept for backward compatibility with previous version.

RETURNS

N/A

ERRNO Not Available

SEE ALSO **unixDrv**

unixDrv( )

NAME **unixDrv( )** – install UNIX disk driver

SYNOPSIS

```
STATUS unixDrv (void)
```

DESCRIPTION This routine is to cause the UNIX disk driver to be linked in when building VxWorks when **INCLUDE_DOS_DISK** component is included in VxWorks image. Otherwise, it is not necessary to call this routine before using the UNIX disk driver.

This routine is only applicable to VxSim for Solaris.

IMPORTANT NOTE This routine is obsolete, but is kept for backward compatibility.

RETURNS **OK** (always).

ERRNO Not Available

SEE ALSO **unixDrv**

unld( )

NAME **unld( )** – unload an object module by specifying a file name or module ID (shell command)

SYNOPSIS

```
STATUS unld
    (
    void * nameOrId,  /* name or ID of the object module file */
    int    options    /* Options to control behavior */
    )
```

DESCRIPTION This routine unloads the specified object module from the system. The module can be specified by name or by module ID. Unloading does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless the module was loaded using **loadModuleAt( )** with user-specified addresses, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the object module from the system symbol table.

(3) It removes the module descriptor from the module list.

Before any modules are unloaded, all breakpoints in the system are deleted. If you need to keep breakpoints, set the options parameter to **UNLD_KEEP_BREAKPOINTS**. To use this option successfully, no breakpoints can be set in the code that is being unloaded.

This routine is a **shell command**. That is, it is designed to be used only in the shell, and not in code running on the target. In future releases, calling **unld( )** directly from code may not be supported.

Note that using this command with an argument that is neither a module name or an ID can cause unpredictable behavior.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrLib**, **loadLib**, **ld( )**, **reld( )**, the VxWorks programmer guides.

unldByGroup( )

NAME **unldByGroup( )** – unload an object module by specifying a group number

SYNOPSIS

```
STATUS unldByGroup
    (
    UINT16 group,   /* group number to unload */
    int    options  /* options */
    )
```

DESCRIPTION This routine unloads an object module that has a group number matching the *group* parameter.

The *options* parameter may be set to any of the options that are available to the **unldByModuleId( )** API. See its reference for more information.

See the manual entries for **unldLib** for more information on module unloading.

RETURNS **OK**, or **ERROR** if there is a problem.

ERRNO Not Available

SEE ALSO **unldLib**, **symLib**, **unldByModuleId( )**

unldByModuleId( )

NAME

unldByModuleId( ) – unload an object module by specifying a module ID

SYNOPSIS

```
STATUS unldByModuleId
    (
    MODULE_ID moduleId,  /* module ID to unload */
    int       options    /* options */
    )
```

DESCRIPTION

This routine unloads an object module that has a module ID matching the *moduleId* parameter.

Unloading does the following:

(1) Frees the space allocated for the code module segments (text, data, and BSS), unless **loadModuleAt( )** was used to specify the locations where the segments were to be loaded, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the code module from the symbol table.

(3) It removes the code module descriptor, its list of segment descriptors, and its list of section descriptors from the kernel's code module list.

The unloader accepts the following options which may be combined by a binary **OR** (**UNLD_CPLUS_XTOR_AUTO** and **UNLD_CPLUS_XTOR_MANUAL** are mutually exclusive):

UNLD_KEEP_BREAKPOINTS
Before any modules are unloaded, all breakpoints in the system are deleted. If you need to keep breakpoints, set the options parameter to **UNLD_KEEP_BREAKPOINTS**. To use this option safely, there should be no breakpoints set in the code that is being unloaded.

UNLD_FORCE
By default, the unloader does not remove the text sections when they are used by some hooks in the system (see the manual of **unldLib** for the list of hooks). Using **UNLD_FORCE** will force the unloader to remove the sections anyway, at the risk of unpredictable results.

UNLD_CPLUS_XTOR_AUTO
This option specifies that the unloader should call the code module's C++ destructor routines.

UNLD_CPLUS_XTOR_MANUAL
This option prevents the unloader from calling the code module's C++ destructor routines. If using this option, the user should be sure that the destructor routines do not perform the release of any resources back to the system, such as memory or semaphores. Or the caller may first cause any static destructors to be run by using the function **cplusDtors( )**.

RETURNS

OK, or **ERROR** if there is a problem.

ERRNO

Possible errnos set by this routine include:

+ **S_moduleLib_INVALID_MODULE_ID**

For a complete description of the errnos, see the reference documentation for **moduleLib**.

SEE ALSO

unldLib

unldByNameAndPath( )

NAME

unldByNameAndPath( ) – unload an object module by specifying a name and path

SYNOPSIS

```
STATUS unldByNameAndPath
    (
    char * name,    /* name of the object module to unload */
    char * path,    /* path to the object module to unload */
    int    options  /* options */
    )
```

DESCRIPTION

This routine unloads an object module specified by the *name* and *path* parameters. The *name* and *path* correspond to the parameters that were passed to the load routine *when the module was loaded*.

The *options* parameter may be set to any of the options that are available to the **unldByModuleId( )** API. See its reference for more information.

See the manual entries for **unldLib** for more information on module unloading.

EXAMPLES

If the module was loaded using the following name and path:

```
fd = open ("path/to/the/module/to/load/moduleName", O_RDONLY);
moduleLoad (fd, LOAD_GLOBAL_SYMBOLS);
```

then the call to **unldByNameAndPath( )** would be done as:

```
unldByNameAndPath ("moduleName", "path/to/the/module/to/load", 0);
```

The path field should be left empty if the module was loaded without any path specified:

```
fd = open ("moduleName", O_RDONLY);
moduleLoad (fd, LOAD_GLOBAL_SYMBOLS);
unldByNameAndPath ("moduleName", "", 0);
```

RETURNS

OK, or **ERROR** if there is a problem.

ERRNO

Not Available

SEE ALSO

unldLib, **unldByModuleId( )**

unlink( )

NAME

unlink( ) – unlink a file

SYNOPSIS

```
int unlink
    (
    const char *name  /* name of the file to remove */
    )
```

DESCRIPTION

This routine removes a link to a file. It shall remove the link named by *name* and decrease the link count of the file referenced by the link.

RETURNS

OK if successful; **ERROR** otherwise.

ERRNO

SEE ALSO

fsPxLib, **link( )**

unstatShow( )

NAME

unstatShow( ) – display all **AF_LOCAL** sockets

SYNOPSIS

```
void unstatShow (void)
```

DESCRIPTION

This routine displays a list of all **AF_LOCAL** family sockets in a format similar to the UNIX **netstat -f unix** command.

Sample output:

```
AF_LOCAL/COMP protocol sockets
                                                            pending
      address                bytes     high     packets  connections
so#  (self/peer)   State    of data  watermark  dropped   (cur/max)
--------------------------------------------------------------------
  7   0001/NONE  LISTENING        0       N/A       N/A       1/ 20
  9   0001/0002  EXCHANGING  30012     65535        10         N/A
  8   0002/0001  EXCHANGING  20014     20014         0         N/A
 10   0003/NONE  LISTENING        0       N/A       N/A       5/  5
 12   0003/0004  DONE_RECV        0     65535       132         N/A
 11   0004/0003  DONE_SEND       16        16         0         N/A
  4   0405/NONE  LISTENING        0       N/A       N/A       0/  5
  5   0000/NONE  CLOSED           0         0         0         N/A
  6   0000/NONE  CLOSED           0         0         0         N/A
```

so#
Socket identifier, relative to owner's RTP.

address
Socket's and peer's addresses (/comp/socket/0xWXYZ). **peer** is only valid in the following state: **EXCHANGING**, **DONE_RECV**, **DONE_SEND**. **NONE** is printed otherwise.

state

CLOSEDstarting and ending state: no data can flow

LISTENINGsocket is a listening one, cannot be used to transfer data

EXCHANGINGdata can be sent in either direction

DONE_SENDsocket can only receive, shutdown(write) has been called on it

DONE_RECVsocket can only send, shutdown(write) has been done on peer

bytes of data
Amount of data that is pending on the socket, waiting to be received.

high watermark
Largest amount of pending data at one given time.

dropped packets
Number of packets dropped due to lack of space in the receiver's buffer space.

pending connections
Current and maximum number of unaccepted connections on a listening socket, i.e.: (number of **connect( )** calls) - (number of **accept( )** calls). **N/A** is printed if not in the LISTENING state.

RETURNS N/A

ERRNO Not Available

SEE ALSO **unShow**

usrClock( )

NAME **usrClock( )** – user-defined system clock interrupt routine

SYNOPSIS

```
void usrClock (void)
```

DESCRIPTION

This routine is called at interrupt level on each clock interrupt. It is installed by **usrRoot( )** with a **sysClkConnect( )** call. It calls all the other packages that need to know about clock ticks, including the kernel itself.

If the application needs anything to happen at the system clock interrupt level, it can be added to this routine.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

usrConfig

usrFdiskPartCreate( )

NAME

usrFdiskPartCreate( ) – create an FDISK-like partition table on a disk

SYNOPSIS

```
STATUS usrFdiskPartCreate
    (
    CBIO_DEV_ID cDev,   /* device representing the entire disk */
    int         nPart,  /* how many partitions needed, default=1, max=4 */
    int         size1,  /* space percentage for second partition */
    int         size2,  /* space percentage for third partition */
    int         size3   /* space percentage for fourth partition */
    )
```

DESCRIPTION

This function may be used to create a basic PC partition table. Such a partition table is not intended to be compatible with other operating systems; it is intended for disks connected to a VxWorks target, but without the access to a PC which may be used to create the partition table.

This function is capable of creating only one partition table - the MBR, and will not create any Bootable or Extended partitions. Therefore, only 4 partitions are supported.

cDev is a CBIO device handle for an entire disk, e.g. a handle returned by **dcacheDevCreate( )**, or if dpartCbio is used, it can be either the Master partition manager handle, or the one of the 0th partition if the disk does not contain a partition table at all.

The *nPart* argument contains the number of partitions to create. If *nPart* is 0 or 1, a single partition covering the entire disk is created. If *nPart* is between 2 and 4, the arguments *size1*, *size2* and *size3* contain (as integers) the *percentage* of disk space to be assigned to the 2nd, 3rd, and 4th partitions respectively. The first partition (partition 0) will be assigned the remaining space. Thus, the sum of the three sizes should be less than 100.

Partition sizes will be rounded down to be multiple of whole tracks so that partition Cylinder/Head/Track fields will be initialized as well as the LBA fields. Although the CHS fields are written they are not used in VxWorks, and can not be guaranteed to work correctly on other systems.

RETURNS

OK or **ERROR** writing a partition table to disk

ERRNO

Not Available

SEE ALSO

usrFdiskPartLib

usrFdiskPartRead( )

NAME

usrFdiskPartRead( ) – read an FDISK-style partition table

SYNOPSIS

```
STATUS usrFdiskPartRead
    (
    CBIO_DEV_ID        cDev,      /* device from which to read blocks */
    PART_TABLE_ENTRY * pPartTab,  /* table where to fill results */
    int                nPart      /* # of entries in <pPartTable> */
    )
```

DESCRIPTION

This function will read and decode a PC formatted partition table on a disk, and fill the appropriate partition table array with the resulting geometry, which should be used by the dpartCbio partition manager to access a partitioned disk with a shared disk cache.

EXAMPLE

The following example shows how a hard disk which is expected to have up to two partitions might be configured, assuming the physical level initialization resulted in the *blkIoDevId* handle:

```
devCbio = dcacheDevCreate( blkIoDevId, 0, 0x20000, "Hard Disk");
mainDevId = dpartDevCreate( devCbio, 2, usrFdiskPartRead )
dosFsDevCreate(  "/disk0a", dpartPartGet (mainDevId, 0), 0,0,0);
dosFsDevCreate(  "/disk0b", dpartPartGet (mainDevId, 1), 0,0,0);
```

RETURNS

OK or **ERROR** if partition table is corrupt

ERRNO

Not Available

SEE ALSO

usrFdiskPartLib

usrFdiskPartShow( )

NAME

usrFdiskPartShow( ) – parse and display partition data

SYNOPSIS

```
STATUS usrFdiskPartShow
    (
    CBIO_DEV_ID cbio,            /* device CBIO handle */
    block_t     extPartOffset,  /* user should pass zero */
    block_t     currentOffset,  /* user should pass zero */
    int         extPartLevel    /* user should pass zero */
    )
```

DESCRIPTION

This routine is intended to be user callable.

A device dependent partition table show routine, this routine outputs formatted data for all partition table fields for every partition table found on a given disk, starting with the MBR sectors partition table. This code can be removed to reduce code size by undefining: **INCLUDE_PART_SHOW** and rebuilding this library and linking to the new library.

This routine takes three arguments. First, a CBIO pointer (assigned for the entire physical disk) usualy obtained from **dcacheDevCreate( )**. It also takes two block_t type arguments and one signed int. The user shall pass zero in these paramaters.

For example:

```
sp usrFdiskPartShow (pCbio,0,0,0)
```

Developers may use size*arch* to view code size.

RETURNS

OK or **ERROR**

ERRNO

Not Available

SEE ALSO

usrFdiskPartLib

usrFormatTrans( )

NAME

usrFormatTrans( ) – Perform a low-level trans XBD format operation

SYNOPSIS

```
STATUS usrFormatTrans
    (
    char *dev,
    int  overhead,
    int  type
    )
```

DESCRIPTION

This routine formats a trans XBD with the specified parameters.

The *dev* parameter is the path name of the device (which will have any existing file system ejected and a rawFS put on it during formatting). The *overhead* parameter specifies the amount of media to use for uncommitted workspace, in parts-per-thousand. The *type* parameter specifies the type of format to use. The value of this parameter will be either **FORMAT_REGULAR** (0) or **FORMAT_TFFS** (1). **FORMAT_REGULAR** initializes a system with 2 master records at the beginning and end of the disk. **FORMAT_TFFS** initializes a system with the first sector unused, and a master record at the end of the disk.

This routine then waits for the device to re-instantiate as a TRFS device, at which point it is safe to format it for dosFs.

EXAMPLE

usrFormatTrans ("/trans", 100, 0);
This formats the device referred to by "/trans" to use 10% of the disk as workspace, and places master records on the first and last sectors of the disk.

RETURNS

OK, or **ERROR** if an error occurs formatting the device.

ERRNO

Not Available

SEE ALSO

usrTransLib

usrIdeConfig( )

NAME

usrIdeConfig( ) – mount a DOS file system from an IDE hard disk

SYNOPSIS

```
STATUS usrIdeConfig
    (
    int    drive,    /* drive number of hard disk (0 or 1) */
    char * fileName  /* mount point */
    )
```

DESCRIPTION

This routine mounts a DOS file system from an IDE hard disk.

The *drive* parameter is the drive number of the hard disk; 0 is **C:** and 1 is **D:**.

The *fileName* parameter is the mount point, e.g., **/ide0/**.

NOTE

Because VxWorks does not support partitioning, hard disks formatted and initialized on VxWorks are not compatible with DOS machines. This routine does not refuse to mount a hard disk that was initialized on VxWorks. The hard disk is assumed to have only one partition with a partition record in sector 0.

RETURNS

OK or **ERROR**.

ERRNO Not Available

SEE ALSO **usrIde**, the VxWorks programmer guides, the architecture supplement.

usrInit( )

NAME **usrInit( )** – user-defined system initialization routine

SYNOPSIS

```
void usrInit
    (
    int startType
    )
```

DESCRIPTION This is the first C code executed after the system boots. This routine is called by the assembly language start-up routine **sysInit( )** which is in the **sysALib** module of the target-specific directory. It is called with interrupts locked out. The kernel is not multitasking at this point.

This routine starts by clearing BSS; thus all variables are initialized to 0, as per the C specification. It then initializes the hardware by calling **sysHwInit( )**, sets up the interrupt/exception vectors, and starts kernel multitasking with **usrRoot( )** as the root task.

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrConfig**, **kernelLib**

usrRoot( )

NAME **usrRoot( )** – the root task

SYNOPSIS

```
void usrRoot
    (
    char *   pMemPoolStart,  /* start of system memory partition */
    unsigned memPoolSize     /* initial size of mem pool */
    )
```

DESCRIPTION This is the first task to run under the multitasking kernel. It performs all final initialization and then starts other tasks.

It initializes the I/O system, installs drivers, creates devices, and sets up the network, etc., as necessary for a particular configuration. It may also create and load the system symbol table, if one is to be included. It may then load and spawn additional tasks as needed. In the default configuration, it simply initializes the VxWorks shell.

RETURNS N/A

ERRNO Not Available

SEE ALSO **usrConfig**

usrScsiConfig( )

NAME **usrScsiConfig( )** – configure SCSI peripherals

SYNOPSIS

```
STATUS usrScsiConfig (void)
```

DESCRIPTION This code configures the SCSI disks and other peripherals on a SCSI controller chain.

The macro **SCSI_AUTO_CONFIG** will include code to scan all possible device/lun id's and to configure a scsiPhysDev structure for each device found. Of course this doesn't include final configuration for disk partitions, floppy configuration parameters, or tape system setup. All of these actions must be performed by user code, either through **sysScsiConfig( )**, the startup script, or by the application program.

The user may customize this code on a per BSP basis using the **SYS_SCSI_CONFIG** macro. If defined, then this routine will call the routine **sysScsiConfig( )**. That routine is to be provided by the BSP, either in **sysLib.c** or **sysScsi.c**. If **SYS_SCSI_CONFIG** is not defined, then **sysScsiConfig( )** will not be called as part of this routine.

An example **sysScsiConfig( )** routine can be found in **target/src/config/usrScsi.c**. The example code contains sample configurations for a hard disk, a floppy disk and a tape unit.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **usrScsi**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

usrTransCommit( )

NAME
usrTransCommit( ) – Set a transaction point on a trans XBD

SYNOPSIS
```
STATUS usrTransCommit
    (
    char *volume
    )
```

DESCRIPTION
This routine sets a transaction point using the volume name of the device.

The *volume* parameter is the name of the device on which TRFS is instantiated.

RETURNS
OK, or **ERROR** if the transaction point is not set.

ERRNO
Not Available

SEE ALSO
usrTransLib

usrTransCommitFd( )

NAME
usrTransCommitFd( ) – set a transaction point using a file descriptor

SYNOPSIS
```
STATUS usrTransCommitFd
    (
    int fd
    )
```

DESCRIPTION
This routine sets a transaction point on the device which contains the filesystem containing the file to which the parameter refers.

The *fd* parameter is a file descriptor, which must refer to a file whose backing media uses a TRFS XBD.

RETURNS
OK, or **ERROR** if the transaction point is not set.

ERRNO
Not Available

SEE ALSO
usrTransLib

uswab( )

NAME **uswab( )** – swap bytes with buffers that are not necessarily aligned

SYNOPSIS

```
void uswab
    (
    char *source,       /* pointer to source buffer      */
    char *destination,  /* pointer to destination buffer */
    int  nbytes         /* number of bytes to exchange   */
    )
```

DESCRIPTION This routine gets the specified number of bytes from *source*, exchanges the adjacent even and odd bytes, and puts them in *destination*.

NOTE: Due to speed considerations, this routine should only be used when absolutely necessary. Use **swab( )** for aligned swaps.

The value of *nBytes* must not be odd. Failure to adhere to this may yield incorrect results.

RETURNS N/A

ERRNO N/A

SEE ALSO **bLib**, **swab( )**

utf16ToCP( )

NAME **utf16ToCP( )** – Convert a UTF-16 encoded Unicode character to a codepoint.

SYNOPSIS

```
int utf16ToCP
    (
    const unsigned short * utf16,
    const int              length,        /* length is in 16-bit words */
    const int              littleEndian,
    unsigned long *        codePoint
    )
```

DESCRIPTION This routine converts a character encoded as UTF-16 to an unsigned long which represents the value of a Unicode characters codepoint.

RETURNS If positive, the return value is the number of UTF-16 words converted. If negative, the value **UC_NOSRC** indicates that insufficient words were given to represent a codepoint. The value **UC_FORMAT** indicates that the UTF-16 vector was of an invalid format.

ERRNO Not Available

SEE ALSO **utfLib**

utf16ToUtf8String( )

NAME **utf16ToUtf8String( )** – Convert a UTF-16 string to a UTF-8 String

SYNOPSIS

```
int utf16ToUtf8String
    (
    const unsigned short * utf16,
    int                    littleEndian,
    unsigned char *        utf8,
    const int              len8
    )
```

DESCRIPTION This routine converts a Zero terminated, UTF-16 encoded string of the indicated endianess to a **NULL** terminated UTF-8 encoded string.

RETURNS If positive, returns the number of bytes used by the resulting UTF-8 encoded string. If non-positive, **UC_FORMAT** indicates that the UTF-16 string is of an invalid format; **UC_BUFFER** indicates that the buffer provided for the UTF-8 string is too small.

ERRNO Not Available

SEE ALSO **utfLib**

utf16ToUtf8StringBOM( )

NAME **utf16ToUtf8StringBOM( )** – Convert UTF-16 to UTF-8 based on a Byte Order Mark

SYNOPSIS

```
int utf16ToUtf8StringBOM
    (
    const unsigned short * utf16,
    unsigned char *        utf8,
    const int              len8
    )
```

DESCRIPTION This routine handles UTF-16 in its standard form. If the first word is a Byte Order Mark - Code Point 0xFEFF, then it is examined for endianness, and the rest of the string is interpreted accordingly. If there is no Byte Order Mark, then the string is interpreted as big-endian representation. Note that the Byte Order Mark is a legitimate, though deprecated, character.

RETURNS If positive, returns the number of bytes used by the resulting UTF-8 encoded string. If non-positive, **UC_FORMAT** indicates that the UTF-16 string is of an invalid format; **UC_BUFFER** indicates that the buffer provided for the UTF-8 string is too small.

ERRNO Not Available

SEE ALSO **utfLib**

utf8ToCP( )

NAME **utf8ToCP( )** – Convert a UTF-8 encoded Unicode character to the Unicode codepoint.

SYNOPSIS

```
int utf8ToCP
    (
    const unsigned char * utf8,
    const int             length,
    unsigned long *       codePoint
    )
```

DESCRIPTION This routine converts UTF-8 to an unsigned long which represents the value of the Unicode codepoint.

RETURNS If positive, the return value is the number of characters converted to this codepoint. If non-positive, the return value of **UC_NOSRC** indicates that there are insufficient characters for a valid conversion, and a return value of **UC_FORMAT** indicates that the format of the input string is not valid UTF-8.

ERRNO Not Available

SEE ALSO **utfLib**

utf8ToUtf16String( )

NAME **utf8ToUtf16String( )** – convert a UTF-8 string to a UTF-16 string

SYNOPSIS

```
int utf8ToUtf16String
```

```
    (
    const unsigned char * utf8,
    unsigned short *      utf16,
    const int             len16,
    int                   littleEndian
    )
```

DESCRIPTION This routine converts a **NULL** terminated UTF-8 encoded string to a ZERO terminated UTF-16 string of the indicated endianess. It does not prepend a Byte Order Marker to the beginning of the string - this must be done before conversion if it is required.

RETURNS If positive the number of 16-bit words actually converted, including the terminating Zero. If non-positive, **UC_FORMAT** indicates that the UTF-8 string is not of a legal format, and **UC_BUFFER** indicates that the provided buffer for containing the UTF-16 string is too small to perform the conversion

ERRNO Not Available

SEE ALSO **utfLib**

utf8ToUtf16StringBOM( )

NAME **utf8ToUtf16StringBOM( )** – Convert UTF-8 to UTF16 with a Byte Order Mark

SYNOPSIS

```
int utf8ToUtf16StringBOM
    (
    const unsigned char * utf8,
    unsigned short *      utf16,
    const int             len16,
    int                   littleEndian
    )
```

DESCRIPTION This routine first writes out the Unicode Byte Order Mark Character, and then converts the UTF-8 encoded string to UTF-16 based on the given endianness

RETURNS If positive the number of 16-bit words actually converted, including the terminating Zero and the Byte Order Marker . If non-positive, **UC_FORMAT** indicates that the UTF-8 string is not of a legal format, and **UC_BUFFER** indicates that the provided buffer for containing the UTF-16 string is too small to perform the conversion.

ERRNO Not Available

SEE ALSO **utfLib**

utfLibInit( )

NAME **utfLibInit( )** – initialize the UTF library

SYNOPSIS

```
void utfLibInit (void)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **utfLib**

utflen16( )

NAME **utflen16( )** – Return the number of 16-bit words used by a UTF-16 encoding.

SYNOPSIS

```
int utflen16
    (
    const unsigned short * utf16
    )
```

DESCRIPTION This routine returns the number of 16-bit words, including the terminating Zero, used by a Zero terminated UTF-16 encoded string.

RETURNS Then number of 16-bit words utilized by a UTF-16 encoding.

ERRNO Not Available

SEE ALSO **utfLib**

utflen8( )

NAME **utflen8( )** – return the encoding length of a **NULL** terminated UTF-8 string

SYNOPSIS

```
int utflen8
```

```
    (
    const unsigned char * utf8
    )
```

DESCRIPTION This routine returns the length occupied by the encoding, as opposed to the number of Unicode characters actually encoded, including the terminating **NULL**

RETURNS The total number of chars up to and including the terminating **NULL**.

ERRNO Not Available

SEE ALSO **utfLib**

utime( )

NAME **utime( )** – update time on a file

SYNOPSIS

```
int utime
    (
    const char *           file,
    const struct utimbuf * newTimes
    )
```

DESCRIPTION Update the timestamp on a file. For filesystems that support this command, the timestamp of the file is updated to the current time.

RETURNS **OK** or **ERROR**.

ERRNO N/A

SEE ALSO **dirLib**, **stat( )**, **fstat( )**, **ls( )**

valloc( )

NAME **valloc( )** – allocate memory on a page boundary from the kernel heap

SYNOPSIS

```
void * valloc
    (
    unsigned size  /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a buffer of *size* bytes from the system memory partition (kernel heap). Additionally, it insures that the allocated buffer begins on a page boundary. Page sizes are architecture-dependent.

RETURNS

A pointer to the newly allocated block, or **NULL** if the buffer could not be allocated or the memory management unit (MMU) support library has not been initialized.

ERRNO

S_memLib_PAGE_SIZE_UNAVAILABLE
Could not obtain the size of a virtual page. Possible error is that virtual memory support is not included (**INCLUDE_MMU_BASIC**).

S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO

memLib

version( )

NAME

version( ) – print VxWorks version information

SYNOPSIS

```
void version (void)
```

DESCRIPTION

This command prints the VxWorks version number, the date this copy of VxWorks was made, and other pertinent information.

EXAMPLE

```
-> version
VxWorks (for SunOS 5.8 [sun4u]) version 6.0.
Kernel: WIND version 2.7.
Made on May 13 2004, 13:23:14.
Boot line:
passDev(0,0)river:/wind/river/target/proj/solaris_diab/default/vxWorks u=user
tn=vxTarget
value = 0 = 0x0
```

RETURNS

N/A

ERRNO

N/A

SEE ALSO

usrLib, the VxWorks programmer guides.

vfdprintf( )

NAME

vfdprintf( ) – write a string formatted with a variable argument list to a file descriptor

SYNOPSIS

```
int vfdprintf
    (
    int          fd,     /* file descriptor to print to */
    const char * fmt,    /* format string for print */
    va_list      vaList  /* optional arguments to format */
    )
```

DESCRIPTION

This routine prints a string formatted with a variable argument list to a specified file descriptor. It is identical to **fdprintf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

The number of characters output, or **ERROR** if there is an error during output.

ERRNO

Not Available

SEE ALSO

fioLib, **fdprintf( )**

virtualDiskClose( )

NAME

virtualDiskClose( ) – close a virtual disk block device.

SYNOPSIS

```
STATUS virtualDiskClose
    (
    BLK_DEV * blkDev  /* virtual disk block device */
    )
```

DESCRIPTION

This routine closes a virtual disk block device by closing the host file associated with the virtual disk.

The *blkDev* parameter specifies the virtual disk to close.

RETURNS

OK on success, else **ERROR**.

ERRNO

Not Available

SEE ALSO

virtualDiskLib

virtualDiskCreate( )

NAME

virtualDiskCreate( ) – create a virtual disk device.

SYNOPSIS

```
BLK_DEV * virtualDiskCreate
    (
    char * hostFile,      /* name of the host file               */
    int    bytesPerBlk,   /* number of bytes per block           */
    int    blksPerTrack,  /* number of blocks per track          */
    int    nBlocks        /* number of blocks on this device     */
    )
```

DESCRIPTION

This routine creates a virtual disk device. The host file is created if it does not exist. If it already exists, only the *hostFile* parameter is taken in account, the others parameters are extracted from the host file.

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 \ 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 disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (i.e., the disk is 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.

RETURNS

A pointer to block device (**BLK_DEV**) structure, or **NULL** if the virtual disk creation failed.

ERRNO

Not Available

SEE ALSO

virtualDiskLib

virtualDiskInit( )

NAME

virtualDiskInit( ) – install the virtual disk driver

SYNOPSIS

```
STATUS virtualDiskInit (void)
```

DESCRIPTION

This routine is used to initialize the virtual disk driver. This routine is automatically called when the **INCLUDE_VIRTUAL_DISK** component is included.

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **virtualDiskLib**

vmArch32LibInit( )

NAME **vmArch32LibInit( )** – initialize the arch specific unbundled VM library (VxVMI Option)

SYNOPSIS

```
void vmArch32LibInit (void)
```

DESCRIPTION This routine links the arch specific unbundled VM library into the VxWorks system. It is called automatically when **INCLUDE_MMU_FULL** and **INCLUDE_MMU_P6_32BIT** are both defined in the BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vmArch32Lib**

vmArch32Map( )

NAME **vmArch32Map( )** – map 32bit physical space into 32bit virtual space (VxVMI Option)

SYNOPSIS

```
STATUS vmArch32Map
    (
    VM_CONTEXT_ID context,    /* context - NULL == currentContext */
    void *        virtAddr,   /* virtual address */
    void *        physAddr,   /* physical address */
    UINT32        stateMask,  /* state mask */
    UINT32        state,      /* state */
    UINT32        len         /* len of virtual and physical spaces */
    )
```

DESCRIPTION vmArch32Map maps 32bit physical pages into a contiguous block of 32bit virtual memory. *virtAddr* and *physAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the mapping the specified state is set to all pages in the newly mapped virtual memory.

The **vmArch32Map( )** routine can fail if the specified virtual address space conflicts with the translation tables of the global virtual memory space. The global virtual address space is initialized at boot time. If a conflict results, **errno** is set to **S_vmLib_ADDR_IN_GLOBAL_SPACE**. To avoid this conflict, use **vmGlobalInfoGet( )** to ascertain which portions of the virtual address space are reserved for the global virtual address space. If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if *virtAddr* or *physAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**
S_vmLib_ADDR_IN_GLOBAL_SPACE

SEE ALSO **vmArch32Lib**

vmArch32Translate( )

NAME **vmArch32Translate( )** – translate a 32bit virtual address to a 32bit physical address (VxVMI Option)

SYNOPSIS

```
STATUS vmArch32Translate
    (
    VM_CONTEXT_ID context,   /* context - NULL == currentContext */
    void *        virtAddr,  /* virtual address */
    void **       physAddr   /* place to put result */
    )
```

DESCRIPTION vmArch32Translate retrieves mapping information for a 32bit virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status is **ERROR**. If *context* is specified as **NULL**, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the validation or translation fails.

ERRNO Not Available

SEE ALSO **vmArch32Lib**

vmArch36LibInit( )

NAME **vmArch36LibInit( )** – initialize the arch specific unbundled VM library (VxVMI Option)

SYNOPSIS

```
void vmArch36LibInit (void)
```

DESCRIPTION This routine links the arch specific unbundled VM library into the VxWorks system. It is called automatically when **INCLUDE_MMU_FULL** and **INCLUDE_MMU_P6_36BIT** are both defined in the BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vmArch36Lib**

vmArch36Map( )

NAME **vmArch36Map( )** – map 36bit physical space into 32bit virtual space (VxVMI Option)

SYNOPSIS

```
STATUS vmArch36Map
    (
    VM_CONTEXT_ID context,    /* context - NULL == currentContext */
    void *        virtAddr,   /* 32bit virtual address */
    LL_INT        physAddr,   /* 36bit physical address */
    UINT32        stateMask,  /* state mask */
    UINT32        state,      /* state */
    UINT32        len         /* len of virtual and physical spaces */
    )
```

DESCRIPTION vmArch36Map maps 36bit physical pages into a contiguous block of 32bit virtual memory. *virtAddr* and *physAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the mapping the specified state is set to all pages in the newly mapped virtual memory.

The **vmArch36Map( )** routine can fail if the specified virtual address space conflicts with the translation tables of the global virtual memory space. The global virtual address space is initialized at boot time. If a conflict results, **errno** is set to **S_vmLib_ADDR_IN_GLOBAL_SPACE**. To avoid this conflict, use **vmGlobalInfoGet( )** to ascertain which portions of the virtual address space are reserved for the global virtual address space. If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if *virtAddr* or *physAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**
S_vmLib_ADDR_IN_GLOBAL_SPACE

SEE ALSO **vmArch36Lib**

vmArch36Translate( )

NAME **vmArch36Translate( )** – translate a 32bit virtual address to a 36bit physical address (VxVMI Option)

SYNOPSIS

```
STATUS vmArch36Translate
    (
    VM_CONTEXT_ID context,   /* context - NULL == currentContext */
    void *        virtAddr,  /* 32bit virtual address */
    LL_INT *      physAddr   /* place to put 36bit result */
    )
```

DESCRIPTION vmArch36Translate retrieves mapping information for a 32bit virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status is **ERROR**. If *context* is specified as **NULL**, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the validation or translation fails.

ERRNO Not Available

SEE ALSO **vmArch36Lib**

vmAttrShow( )

NAME

vmAttrShow( ) – display the text representation of a MMU attribute value

SYNOPSIS

```
STATUS vmAttrShow
    (
    UINT pgAttr,      /* MMU attributes */
    UINT pgAttrMask   /* MMU attributes mask */
    )
```

DESCRIPTION

This routine will display the text value of the attributes passed in *pgAttr*. This information includes the supervisor and user RWX values, cache mode (CB/WT/OFF), coherency, and I/O settings.

Note that this routine cannot report non-standard architecture-dependent states.

RETURNS

ERROR if *pgAttrMask* is not a valid mask or **NULL**, else **OK**.

ERRNO

Not Available

SEE ALSO

vmShow

vmBaseArch32LibInit( )

NAME

vmBaseArch32LibInit( ) – initialize the arch specific bundled VM library

SYNOPSIS

```
void vmBaseArch32LibInit (void)
```

DESCRIPTION

This routine links the arch specific bundled VM library into the VxWorks system. It is called automatically when **INCLUDE_MMU_BASIC** and **INCLUDE_MMU_P6_32BIT** are both defined in the BSP.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

vmBaseArch32Lib

vmBaseArch32Map( )

NAME

vmBaseArch32Map( ) – map 32bit physical to the 32bit virtual memory

SYNOPSIS

```
STATUS vmBaseArch32Map
    (
    void * virtAddr,   /* 32bit virtual address */
    void * physAddr,   /* 32bit physical address */
    UINT32 stateMask,  /* state mask */
    UINT32 state,      /* state */
    UINT32 len         /* length */
    )
```

DESCRIPTION

vmBaseArch32Map maps 32bit physical pages into a contiguous block of 32bit virtual memory. *virtAddr* and *physAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the mapping the specified state is set to all pages in the newly mapped virtual memory.

This routine should not be called from interrupt level.

RETURNS

OK, or **ERROR** if *virtAddr* or *physAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED

SEE ALSO

vmBaseArch32Lib

vmBaseArch32Translate( )

NAME

vmBaseArch32Translate( ) – translate a 32bit virtual address to a 32bit physical address

SYNOPSIS

```
STATUS vmBaseArch32Translate
    (
    void *  virtAddr,  /* virtual address */
    void ** physAddr   /* place to put result */
    )
```

DESCRIPTION

vmBaseArch32Translate retrieves mapping information for a 32bit virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status is **ERROR**.

This routine is callable from interrupt level.

RETURNS

OK, or **ERROR** if validation or translation fails.

ERRNO Not Available

SEE ALSO **vmBaseArch32Lib**

vmBaseArch36LibInit( )

NAME **vmBaseArch36LibInit( )** – initialize the arch specific bundled VM library

SYNOPSIS

```
void vmBaseArch36LibInit (void)
```

DESCRIPTION This routine links the arch specific bundled VM library into the VxWorks system. It is called automatically when **INCLUDE_MMU_BASIC** and **INCLUDE_MMU_P6_36BIT** are both defined in the BSP.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vmBaseArch36Lib**

vmBaseArch36Map( )

NAME **vmBaseArch36Map( )** – map 36bit physical to the 32bit virtual memory

SYNOPSIS

```
STATUS vmBaseArch36Map
    (
    void * virtAddr,    /* 32bit virtual address */
    LL_INT physAddr,    /* 36bit physical address */
    UINT32 stateMask,   /* state mask */
    UINT32 state,       /* state */
    UINT32 len          /* length */
    )
```

DESCRIPTION vmBaseArch36Map maps 36bit physical pages into a contiguous block of 32bit virtual memory. *virtAddr* and *physAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the mapping the specified state is set to all pages in the newly mapped virtual memory.

This routine should not be called from interrupt level.

RETURNS **OK**, or **ERROR** if *virtAddr* or *physAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**

SEE ALSO **vmBaseArch36Lib**

vmBaseArch36Translate( )

NAME **vmBaseArch36Translate( )** – translate a 32bit virtual address to a 36bit physical address

SYNOPSIS

```
STATUS vmBaseArch36Translate
    (
    void *   virtAddr,  /* 32bit virtual address */
    LL_INT * physAddr   /* place to put 36bit result */
    )
```

DESCRIPTION vmBaseArch36Translate retrieves mapping information for a 32bit virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status is **ERROR**.

This routine is callable from interrupt level.

RETURNS **OK**, or **ERROR** if validation or translation fails.

ERRNO Not Available

SEE ALSO **vmBaseArch36Lib**

vmBaseGlobalMapInit( )

NAME **vmBaseGlobalMapInit( )** – initialize global mapping (obsolete)

SYNOPSIS

```
VM_CONTEXT_ID vmBaseGlobalMapInit
    (
    PHYS_MEM_DESC *pMemDescArray,        /* pointer to array of mem descs */
    int           numDescArrayElements,  /* no. of elements in pMemDescArray
*/
    BOOL          enable,                /* enable virtual memory */
    int           cacheDefault           /* default data cache mode */
    )
```

DESCRIPTION This function will be replaced by **vmGlobalMapInit( )**

RETURNS A pointer to a newly created virtual memory context, or **NULL** if memory cannot be mapped.

ERRNO Not Available

SEE ALSO **vmGlobalMap**, **vmBaseLibInit( )**, **vmGlobalMapInit( )**

vmBasePageSizeGet( )

NAME **vmBasePageSizeGet( )** – return the MMU page size (obsolete)

SYNOPSIS

```
int vmBasePageSizeGet (void)
```

DESCRIPTION This routine is to be replaced by **vmPageSizeGet( )**.

RETURNS The MMU page size of the current architecture.

ERRNO Not Available

SEE ALSO **vmBaseLib**, **vmPageSizeGet( )**

vmBaseStateSet( )

NAME **vmBaseStateSet( )** – change the state of a block of virtual memory (obsolete)

SYNOPSIS

```
STATUS vmBaseStateSet
    (
    VM_CONTEXT_ID context,    /* context - NULL == currentContext */
    VIRT_ADDR     virtAdrs,   /* virtual address to modify state of */
    msize_t       len,        /* len of virtual space to modify state of */
    UINT          stateMask,  /* state mask */
    UINT          state       /* state */
    )
```

DESCRIPTION This function will be replaced by **vmStateSet( )**.

RETURNS **OK**, or **ERROR** if the validation fails, *virtAdrs* is not on a page boundary, *len* is not a multiple of the page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED
virtualAddr must be aligned on a page boundary.

S_vmLib_BAD_STATE_PARAM
state is not a valid combination of MMU states.

S_vmLib_BAD_MASK_PARAM
stateMask is not a valid combination of MMU state masks.

SEE ALSO

vmBaseLib, **vmStateSet( )**

vmContextShow( )

NAME

vmContextShow( ) – display the translation table for a context

SYNOPSIS

```
STATUS vmContextShow
    (
    VM_CONTEXT_ID context  /* VM context - NULL == currentContext */
    )
```

DESCRIPTION

This routine displays the translation table for a specified context. If *context* is specified as **NULL**, the current context is displayed. Output is formatted to show blocks of virtual memory with consecutive physical addresses and the same state. State information shows the read/write/execute status for both USR and SUP modes as well as the cacheablity. Only virtual memory that has its valid state bit set is displayed.

This routine should be used for debugging purposes only.

EXAMPLE

The following example shows the output of **vmContextShow( )** using the shell's C-interpreter:

```
-> vmContextShow
VIRTUAL ADDR  BLOCK LENGTH  PHYSICAL ADDR  PROT (S/U)  CACHE     SPECIAL
------------  ------------  -------------  ----------  -------   ------------
0x60000000    0x00010000    0x60000000     RWX / ---   CB-/--/-  --
0x60010000    0x0014c000    0x60010000     R-X / ---   CB-/--/-  --
0x6015c000    0x0040e000    0x6015c000     RWX / ---   CB-/--/-  --
0x6056a000    0x00004000    0x6056a000     R-X / ---   CB-/--/G  --
0x6056e000    0x00002000    0x6056e000     RWX / ---   CB-/CO/-  --
0x60570000    0x00001000    0x60570000     RWX / ---   WT-/--/-  --
0x60571000    0x00001000    0x60571000     RWX / ---   OFF/--/-  NB
```

For the command-interpreter shell, use the **vm context** command.

The protection attributes (Read/Write/eXecute) are listed separately for supervisor and user mode (S/U).

Cache attributes are listed with the following notation:

Attribute	Meaning
CB	Copyback
WT	Write-through
OFF	Cache disabled
CO	Coherency enabled
G	Guarded

Special attributes are listed with the following notation:

Attribute	Meaning
NB	No-block. See note blow.
S0-S6	Special attributes 0 to 6. See the Architecure Supplement for usage.

The no-block attribute has meaning only on systems where page optimization is used. For more information see **vmPageOptimize( )**.

AVAILABILITY
This routine is distributed as a component of the bundled virtual memory support option.

RETURNS
OK, or **ERROR** if the virtual memory context is invalid.

ERRNO
Not Available

SEE ALSO
vmShow

vmGlobalMapInit( )

NAME
vmGlobalMapInit( ) – initialize global mapping

SYNOPSIS
```
VM_CONTEXT_ID vmGlobalMapInit
    (
    PHYS_MEM_DESC * pMemDescArray,         /* pointer to array of mem descs
*/
    int             numDescArrayElements,  /* no. of elements in
pMemDescArray */
    BOOL            enable,                /* enable virtual memory */
    int             cacheDefault           /* default data cache mode */
    )
```

DESCRIPTION
This routine creates and installs a virtual memory context with mappings defined for each contiguous memory segment defined in *pMemDescArray*. In the standard VxWorks configuration, an instance of **PHYS_MEM_DESC** (called **sysPhysMemDesc**) is defined in **sysLib.c**; the variable is passed to **vmGlobalMapInit( )** by the system configuration mechanism.

This routine is called only once during system initialization. It should never be called by application code.

If *enable* is **TRUE**, the MMU is enabled upon return.

RETURNS A pointer to a newly created virtual memory context, or **NULL** if memory cannot be mapped.

ERRNO Not Available

SEE ALSO **vmGlobalMap**, **vmBaseLibInit( )**

vmMap( )

NAME **vmMap( )** – map physical space into virtual space

SYNOPSIS

```
STATUS vmMap
    (
    VM_CONTEXT_ID context,       /* context - NULL == currentContext   */
    VIRT_ADDR     virtualAddr,   /* virtual address                    */
    PHYS_ADDR     physicalAddr,  /* physical address                   */
    msize_t       len            /* len of virtual and physical spaces */
    )
```

DESCRIPTION This routine maps physical pages into a contiguous block of virtual memory. *virtualAddr* and *physicalAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the call to **vmMap( )**, the state of all pages in the the newly mapped virtual memory is valid, accessible in SUP mode, and cacheable. Note: If mapping a particular page within the given range fails , then the pages that have already been mapped is not restored back.

If *context* is specified as **NULL**, the current virtual memory context is used.

The physicalAddr has to be of type (**PHYS_ADDR**) since on some architectures the physical address could represent more than 32 bits.

This routine should not be called from interrupt level.

RETURNS **OK**, or **ERROR** if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**
virtualAddr must be aligned on a page boundary.

SEE ALSO **vmBaseLib**

vmPageLock( )

NAME

vmPageLock( ) – lock the pages.

SYNOPSIS

```
STATUS vmPageLock
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext   */
    VIRT_ADDR     virtualAddr,  /* virtual address                   */
    msize_t       len,          /* len of virtual address             */
    UINT          option        /* unused. (for future if needed)     */
    )
```

DESCRIPTION

This routine will lock the pages by using a static TLB entry if possible.

If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

IMPORTANT

The support for this routine is not available on many CPUs and architectures. You should reference your architecture supplement to see if this is available.

Locking of the vxWorks image text section is configurable by using **INCLUDE_LOCK_TEXT_SECTION**.

This routine currently only will lock a valid page present in the kernel context. It will return an **ERROR** if this is not the case. Also if a page is locked, it can no longer can have its state changed; therefore if a call from vmStateSet is made on a locked page it will return an **ERROR**. Finally, there is an additional errno for vmStateSet if a page is locked: **S_mmuLib_TLB_LOCKED_PAGE**.

RETURNS

OK, or **ERROR** if *virtualAddr* is not on page boundaries, *len* is not a multiple of the page size or if the locking of the pages cannot be done.

ERRNO

S_vmLib_FUNCTION_UNSUPPORTED
  page locking function not supported.

S_vmLib_NOT_PAGE_ALIGNED
  virtualAddr must be aligned on a page boundary.

S_mmuLib_TLB_LOCKED_PAGE
  Already a locked page

S_mmuLib_NOT_CONTIGUOUS_ADDR
  Requires contiguous phys addr

S_mmuLib_NOT_CONTIGUOUS_STATE
  Requires contiguous MMU state

S_mmuLib_INVALID_DESCRIPTOR
  Bad address

S_mmuLib_NOT_GLOBAL_PAGE
Page must be shared by all contexts

S_mmuLib_LOCK_NO_MORE_TLB_RESOURCES
No more TLB entries available

SEE ALSO **vmBaseLib**

vmPageMap( )

NAME **vmPageMap( )** – map physical space into virtual space

SYNOPSIS

```
STATUS vmPageMap
    (
    VM_CONTEXT_ID context,       /* context - NULL == currentContext   */
    VIRT_ADDR     virtualAddr,   /* virtual address                    */
    PHYS_ADDR     physicalAddr,  /* physical address                   */
    msize_t       len,           /* len of virtual and physical spaces */
    UINT          stateMask,     /* combination of MMU state masks.    */
    UINT          state          /* combination of MMU states.         */
    )
```

DESCRIPTION This routine maps physical pages into a contiguous block of virtual memory. *virtualAddr* and *physicalAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the call to **vmMap( )**, the state of all pages in the the newly mapped virtual memory is set to the default value (valid, sup rwx & cache default) if stateMask is passed as **NULL** , or else it is set to whatever is passed via statMask/state Note: If mapping a particular page within the given range fails , then the pages that have already been mapped is not restored back.

If *context* is specified as **NULL**, the current virtual memory context is used.

The physicalAddr has to be of type (**PHYS_ADDR**) since on some architectures the physical address could represent more than 32 bits.

This routine should not be called from interrupt level. This routine cannot be called via the macro **VM_PAGE_MAP**.

RETURNS **OK**, or **ERROR** if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails, or if invalid state or stateMask are passed.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**
virtualAddr must be aligned on a page boundary.

SEE ALSO **vmBaseLib**

vmPageOptimize( )

NAME

vmPageOptimize( ) – Optimize the address range if possible.

SYNOPSIS

```
STATUS vmPageOptimize
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext   */
    VIRT_ADDR     virtualAddr,  /* virtual address                    */
    msize_t       len,          /* len of address range in bytes      */
    UINT          option        /* unused. for future if needed       */
    )
```

DESCRIPTION

This routine will try to optimize the passed address range by modifing pages to use MMU page sizes larger than the default one **VM_PAGE_SIZE**, if possible.

If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

The support for this routine is not available on many CPUs and architectures.

IMPORTANT

You should reference your architecture supplement to see if this is available.

WARNING

One side affect after using this routine, when supported, is that vmStateSet will possibly block with a semaphore except in interrupt where it returns **ERROR**. To prevent a call to vmStateSet from returning an **ERROR** when in an ISR you must preempt with a call, for the same address range, to vmStateSet setting the special state **MMU_ATTR_NO_BLOCK**. This should be treated as a special purpose attribute.

There is an additional errno for vmStateSet if optimization is enabled:
S_mmuLib_ISR_CALL_BLOCKED - StateSet needed to block because of optimization so returned error. Should have used **MMU_ATTR_NO_BLOCK** state set on address.

Initial optimization of the whole of kernel context can be done by just configuring in **INCLUDE_PAGE_SIZE_OPTIMIZATION**.

RETURNS

OK, or **ERROR** if *virtualAddr* is not on page boundary, *len* is not a multiple of the page size or if the optimization is not possible.

ERRNO

S_vmLib_FUNCTION_UNSUPPORTED
page optimization function not supported.

S_vmLib_NOT_PAGE_ALIGNED
virtualAddr must be aligned on a page boundary.

SEE ALSO

vmBaseLib

vmPageSizeGet( )

NAME **vmPageSizeGet( )** – return the page size

SYNOPSIS

```
int vmPageSizeGet (void)
```

DESCRIPTION This routine returns the architecture-dependent MMU page size.

This routine is callable from interrupt level.

RETURNS The page size of the current architecture.

ERRNO Not Available

SEE ALSO **vmBaseLib**

vmPageUnlock( )

NAME **vmPageUnlock( )** – unlock the pages.

SYNOPSIS

```
STATUS vmPageUnlock
    (
    VM_CONTEXT_ID context,     /* context - NULL == currentContext   */
    VIRT_ADDR     virtualAddr  /* virtual address                    */
    )
```

DESCRIPTION This routine will lock the pages that were locked by a previous vmPageLock.

If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

RETURNS **OK**, or **ERROR** if *virtualAddr* is not on page boundaries, if the pages were not previously locked.

ERRNO

S_vmLib_FUNCTION_UNSUPPORTED
page unlocking function not supported.

S_vmLib_NOT_PAGE_ALIGNED
virtualAddr must be aligned on a page boundary.

S_mmuLib_TLB_PAGE_NOT_LOCKED
Can only unlock a locked page.

SEE ALSO **vmBaseLib**

vmPhysTranslate( )

NAME **vmPhysTranslate( )** – translate a physical address to a virtual address

SYNOPSIS

```
STATUS vmPhysTranslate
    (
    VM_CONTEXT_ID context,       /* context - NULL == currentContext */
    PHYS_ADDR     physicalAddr,  /* physical address */
    VIRT_ADDR  *  virtualAddr    /* place to put result */
    )
```

DESCRIPTION This routine retrieves mapping information for a physical address from the page translation tables. If *context* is specified as **NULL**, the current context is used.

The physicalAddr has to be of type (**PHYS_ADDR**) since on some architectures the physical address could represent more than 32 bits.

This routine is callable from interrupt level.

RETURNS **OK**, or **ERROR** if the validation or translation failed.

ERRNO Not Available

SEE ALSO **vmBaseLib**

vmStateGet( )

NAME **vmStateGet( )** – get the state of a page of virtual memory

SYNOPSIS

```
STATUS vmStateGet
    (
    VM_CONTEXT_ID context,   /* VM context; use NULL for current context */
    VIRT_ADDR     pageAddr,  /* virtual page addr */
    UINT *        pState     /* where to return state */
    )
```

DESCRIPTION This routine gets the MMU attributes of a page mapped in a virtual memory context. For a description of the supported page attributes see the **vmStateSet( )** API guide.

If *context* is **NULL**, the current virtual memory context is used.

This routine is callable from interrupt level.

For example, to see if a page is writable in supervisor mode, the following code may be used:

```
    if (vmStateGet (context, pageAddr, &attr) == OK)
        {
        if (((attr & MMU_ATTR_VALID_MSK) == MMU_ATTR_VALID) &&
            ((attr & MMU_ATTR_PROT_SUP_WRITE) == MMU_ATTR_PROT_SUP_WRITE))
            ...
```

RETURNS **OK**, or **ERROR** if *pageAddr* is not on a page boundary, the validity check fails, or the architecture-dependent state get fails for the specified virtual address.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**
pageAddr is not aligned on a page boundary.

SEE ALSO **vmBaseLib**, **vmStateSet( )**

vmStateSet( )

NAME **vmStateSet( )** – change the state of a block of virtual memory

SYNOPSIS

```
STATUS vmStateSet
    (
    VM_CONTEXT_ID context,     /* context - NULL == currentContext */
    VIRT_ADDR     virtAdrs,    /* virtual address to modify state of */
    msize_t       len,         /* len of virtual space to modify state of */
    UINT          stateMask,   /* state mask */
    UINT          state        /* state */
    )
```

DESCRIPTION This routine changes the MMU attributes of a block of virtual memory. Each page of virtual memory has at least three types of state information: validity, protection, and cacheability. Some architectures define additional state information.

The following MMU attributes are supported and may be OR'ed together in the *state* parameter:

Protection attributes:

Attribute	Description
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

Validity attribute. Memory accesses to a page set invalid will result in an exception.

Attribute	Description
MMU_ATTR_VALID	page is valid
MMU_ATTR_VALID_NOT	page is not valid

Cache attributes:

Attribute	Description
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_DEFAULT	default cache value, **USER_D_CACHE_MODE**
MMU_ATTR_CACHE_GUARDED	page access set to guarded
MMU_ATTR_CACHE_COHERENCY	page access set to cache coherent

The *stateMask* parameter is used to specify which MMU attribute groups are being modified. This should be an inclusive OR of one or more of the following masks:

Mask	Description
MMU_ATTR_PROT_MSK	set protection attributes
MMU_ATTR_VALID_MSK	set valid attribute
MMU_ATTR_CACHE_MSK	set cache attributes
MMU_ATTR_SPL_MSK	set architecture specific attributes

The following restrictions must be respected when setting page attributes:

- only one of **MMU_ATTR_CACHE_OFF, MMU_ATTR_CACHE_COPYBACK, MMU_ATTR_CACHE_WRITETHRU** or **MMU_ATTR_CACHE_DEFAULT** can be set at any time.
- not all combinations of the protection attributes are supported by various architectures. For more information see the respective Architecture Supplement documentation.

Refer to the archecture specific **mmuLib** man pages for specific details.

If *context* is **NULL**, the current context is used.

This routine is callable from interrupt level.

RETURNS **OK**, or **ERROR** if the validation fails, *virtAdrs* is not on a page boundary, *len* is not a multiple of the page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED
virtAdrs is not aligned on a page boundary.

S_vmLib_BAD_STATE_PARAM
state is not a valid combination of MMU attributes.

S_vmLib_BAD_MASK_PARAM
stateMask is not a valid combination of MMU attribute masks.

SEE ALSO **vmBaseLib**, **vmStateGet( )**

vmTextProtect( )

NAME **vmTextProtect( )** – write-protect kernel text segment

SYNOPSIS

```
STATUS vmTextProtect
    (
    BOOL setState
    )
```

DESCRIPTION This routine enables write-protection of text segments in the VxWorks kernel. This function should not be called by application code; instead, this routine is called automatically at boot time when the **INCLUDE_PROTECT_TEXT** component is included.

If the start of the text segment is not page aligned, text protection starts from the next page boundary. This routine expects that the data segment follows the text segment after a proper alignment padding. The VxWorks build system ensures this condition except for ROM-resident images (i.e. images for which the text segment is directly executed out of ROM or flash memory). For ROM-resident images, if detection of an attempt to write in the ROM is desired, protection should be enabled via the corresponding sysPhysMemDesc[] entry in the BSP.

This routine is not setting protection attributes for the MIPS architecture. Text protection for MIPS, when **INCLUDE_PROTECT_TEXT** is included, is enabled when the initial mappings are created.

RETURNS **OK**, or **ERROR** if the text segment cannot be write-protected.

ERRNO **S_vmLib_TEXT_PROTECTION_UNAVAILABLE**
write-protecting the kernel text segment is not supported.

SEE ALSO **vmBaseLib**

vmTranslate( )

NAME **vmTranslate( )** – translate a virtual address to a physical address

SYNOPSIS

```
STATUS vmTranslate
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext */
    VIRT_ADDR     virtualAddr,  /* virtual address */
    PHYS_ADDR *   physicalAddr  /* place to put result */
    )
```

DESCRIPTION This routine retrieves mapping information for a virtual address from the page translation tables. If *context* is specified as **NULL**, the current context is used.

The physicalAddr has to be of type (**PHYS_ADDR ***) since on some architectures the physical address could represent more than 32 bits.

This routine is callable from interrupt level.

RETURNS **OK**, or **ERROR** if the validation or translation failed.

ERRNO Not Available

SEE ALSO **vmBaseLib**

voprintf( )

NAME **voprintf( )** – write a formatted string to an output function

SYNOPSIS

```
int voprintf
    (
    FUNCPTR      prtFunc,  /* pointer to output function */
    int          prtArg,   /* argument for output function */
    const char * fmt,      /* format string to write */
    va_list      vaList    /* optional arguments to format */
    )
```

DESCRIPTION This routine prints a formatted string via the function specified by *prtFunc*. The function will receive as parameters a pointer to a buffer, an integer indicating the length of the buffer, and the argument *prtArg*. If **NULL** is specified as the output function, the output will be sent to stdout.

This routine is identical to **oprintf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.\

RETURNS The number of characters output, not including the **NULL** terminator.

ERRNO Not Available

SEE ALSO **fioLib**, **oprintf( )**, **printf( )**

vprintf( )

NAME **vprintf( )** – write a string formatted with a variable argument list to standard output (ANSI)

SYNOPSIS

```
int vprintf
    (
    const char * fmt,    /* format string to write */
    va_list      vaList  /* arguments to format */
    )
```

DESCRIPTION This routine prints a string formatted with a variable argument list to standard output. It is identical to **printf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

ERRNO Not Available

SEE ALSO **fioLib**, **printf( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output (***stdio.h***)*

vrfsDevCreate( )

NAME **vrfsDevCreate( )** – Instantiate the VRFS

SYNOPSIS

```
STATUS vrfsDevCreate
    (
    )
```

DESCRIPTION This routine creates an instance of the VRFS if one does not exist. That instance will always be installed as device "/".

RETURNS **OK** on success, **ERROR** if failure

ERRNO **EEXIST** if this FS is already instantiated.

SEE ALSO **vrfsLib**

vrfsInit( )

NAME **vrfsInit( )** – Initialize the Virtual Root File System Library

SYNOPSIS

```
STATUS vrfsInit(void)
```

DESCRIPTION This routine initializes the Virtual Root File System. It should be called only once, and initializes the vrfs Core IO driver as well as data structures for the library.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vrfsLib**

vsnprintf( )

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

SYNOPSIS

```
int vsnprintf
    (
    char *       buffer,  /* buffer to write to */
    size_t       count,   /* max number of characters to store in buffer */
    const char * fmt,     /* format string */
    va_list      vaList   /* optional arguments to format */
    )
```

DESCRIPTION This routine copies a string formatted with a variable argument list to a specified buffer, up to a given number of characters. The formatted string will be null terminated. This routine guarantees never to write beyond the provided buffer regardless of the format specifier or the arguments to be formatted. The *count* argument specifies the maximum number of characters to store in the buffer, including the null terminator.

This routine is identical to **snprintf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

RETURNS The number of characters copied to *buffer*, not including the **NULL** terminator.

Even when the supplied *buffer* is too small to hold the complete formatted string, the return value represents the number of characters that would have been written to *buffer* if *count* was sufficiently large.

ERRNO
Not Available

SEE ALSO
fioLib, **sprintf( )**, **printf( )**, *"International Organization for Standardization, ISO/IEC 9899:1999, Programming languages - C: Input/output (***stdio.h***)"*

vsprintf( )

NAME
vsprintf( ) – write a string formatted with a variable argument list to a buffer (ANSI)

SYNOPSIS

```
int vsprintf
    (
    char *       buffer,  /* buffer to write to */
    const char * fmt,     /* format string */
    va_list      vaList   /* optional arguments to format */
    )
```

DESCRIPTION
This routine copies a string formatted with a variable argument list to a specified buffer. This routine is identical to **sprintf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

RETURNS
The number of characters copied to *buffer*, not including the **NULL** terminator.

ERRNO
Not Available

SEE ALSO
fioLib, **sprintf( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: Input/Output (***stdio.h***)*

vxAtomicAdd( )

NAME
vxAtomicAdd( ) – atomically add a value to a memory location

SYNOPSIS

```
atomicVal_t vxAtomicAdd
    (
    atomic_t *  target,  /* memory location to add to */
    atomicVal_t value    /* value to add */
    )
```

DESCRIPTION This routine atomically adds **target* and *value*, placing the result in **target*. The operation is done using signed integer arithmetic. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxAtomicAnd( )

NAME **vxAtomicAnd( )** – atomically perform a bitwise AND on a memory location

SYNOPSIS

```
atomicVal_t vxAtomicAnd
    (
    atomic_t *  target,  /* memory location to AND */
    atomicVal_t value    /* AND with this value */
    )
```

DESCRIPTION This routine atomically performs a bitwise AND operation of **target* and *value*, placing the result in **target*. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxAtomicClear( )

NAME **vxAtomicClear( )** – atomically clear a memory location

SYNOPSIS

```
atomicVal_t vxAtomicClear
```

```
    (
    atomic_t * target  /* memory location to clear */
    )
```

DESCRIPTION This routine atomically clears **target* and returns the old value that was in **target*. Note that all CPU architectures supported by VxWorks can atomically clear a variable of size atomic_t without the need to use this routine. This routine is intended for software that needs to atomically fetch and clear the value of a memory location. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxAtomicDec( )

NAME **vxAtomicDec( )** – atomically decrement a memory location

SYNOPSIS

```
atomicVal_t vxAtomicDec
    (
    atomic_t * target  /* memory location to decrement */
    )
```

DESCRIPTION This routine atomically decrements the value in **target*. The operation is done using unsigned integer arithmetic. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxAtomicGet( )

NAME

vxAtomicGet( ) – atomically get a memory location

SYNOPSIS

```
atomicVal_t vxAtomicGet
    (
    atomic_t * target  /* memory location to get */
    )
```

DESCRIPTION

This routine atomically reads **target* and returns the value. This routine is intended for software that needs to atomically fetch and replace the value of a memory location.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target*.

ERRNO

N/A

SEE ALSO

vxAtomicLib

vxAtomicInc( )

NAME

vxAtomicInc( ) – atomically increment a memory location

SYNOPSIS

```
atomicVal_t vxAtomicInc
    (
    atomic_t * target  /* memory location to increment */
    )
```

DESCRIPTION

This routine atomically increments the value in **target*. The operation is done using unsigned integer arithmetic. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target* before the atomic operation

ERRNO

N/A

SEE ALSO

vxAtomicLib

vxAtomicNand( )

NAME

vxAtomicNand( ) – atomically perform a bitwise NAND on a memory location

SYNOPSIS

```
atomicVal_t vxAtomicNand
    (
    atomic_t *  target,  /* memory location to NAND */
    atomicVal_t value    /* NAND with this value */
    )
```

DESCRIPTION

This routine atomically performs a bitwise NAND operation of **target* and *value*, placing the result in **target*. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target* before the atomic operation

ERRNO

N/A

SEE ALSO

vxAtomicLib

vxAtomicOr( )

NAME

vxAtomicOr( ) – atomically perform a bitwise OR on memory location

SYNOPSIS

```
atomicVal_t vxAtomicOr
    (
    atomic_t *  target,  /* memory location to OR */
    atomicVal_t value    /* OR with this value */
    )
```

DESCRIPTION

This routine atomically performs a bitwise OR operation of **target* and *value*, placing the result in **target*. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target* before the atomic operation

ERRNO

N/A

SEE ALSO

vxAtomicLib

vxAtomicSet( )

NAME

vxAtomicSet( ) – atomically set a memory location

SYNOPSIS

```
atomicVal_t vxAtomicSet
    (
    atomic_t *  target,  /* memory location to set */
    atomicVal_t value    /* set with this value */
    )
```

DESCRIPTION

This routine atomically sets **target* to *value* and returns the old value that was in **target*. Note that all CPU architectures supported by VxWorks can atomically write to a variable of size atomic_t without the need to use this routine. This routine is intended for software that needs to atomically fetch and replace the value of a memory location. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target* before the atomic operation

ERRNO

N/A

SEE ALSO

vxAtomicLib

vxAtomicSub( )

NAME

vxAtomicSub( ) – atomically subtract a value from a memory location

SYNOPSIS

```
atomicVal_t vxAtomicSub
    (
    atomic_t *  target,  /* memory location to subtract from */
    atomicVal_t value    /* value to sub */
    )
```

DESCRIPTION

This routine atomically subtracts *value* from **target*, placing the result in **target*. The operation is done using signed integer arithmetic. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS

Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxAtomicXor( )

NAME **vxAtomicXor( )** – atomically perform a bitwise XOR on a memory location

SYNOPSIS

```
atomicVal_t vxAtomicXor
    (
    atomic_t *  target,  /* memory location to XOR */
    atomicVal_t value    /* XOR with this value */
    )
```

DESCRIPTION This routine atomically performs a bitwise XOR operation of **target* and *value*, placing the result in **target*. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS Contents of **target* before the atomic operation

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxCas( )

NAME **vxCas( )** – atomically compare-and-swap the contents of a memory location

SYNOPSIS

```
BOOL vxCas
    (
    atomic_t *  target,     /* memory location to compare-and-swap */
    atomicVal_t oldValue,   /* compare to this value */
    atomicVal_t newValue    /* swap with this value */
    )
```

DESCRIPTION This routine performs an atomic compare-and-swap; testing that **target* contains *oldValue*, and if it does, setting the value of **target* to *newValue*. Various CPU architectures may impose restrictions with regards to the alignment and cache attributes of the atomic_t type.

This routine can be used from both task and interrupt level.

RETURNS **TRUE** if the swap is actually executed, **FALSE** otherwise.

ERRNO N/A

SEE ALSO **vxAtomicLib**

vxCpuConfiguredGet( )

NAME **vxCpuConfiguredGet( )** – get the number of configured CPUs in the system

SYNOPSIS

```
unsigned int vxCpuConfiguredGet (void)
```

DESCRIPTION This routine returns the number of CPUs that have been configured in the SMP system, whether they have been enabled or not. This number is set at compile time and stays constant for as long as the system is up and running. This routine can therefore be called at any time, even during the booting sequence of the system. Its purpose is to assist initialization code of a kernel application in determining how many per-CPU objects would need to be allocated in an SMP system.

This routine exists because VxWorks SMP has the flexibility to allow the number of CPUs configured in a VxWorks SMP system to be different than the number of available CPUs on the hardware platform. For example, it would be possible to dedicate two cores of a quad-core platform to run VxWorks SMP while the other two cores are used for another purpose.

Calling this routine in the uniprocessor version of VxWorks returns 1, always. This routine can be called from both task an interrupt level.

RETURNS The number of CPUs configured in the system.

ERRNO N/A

SEE ALSO **vxCpuLib**, **vxCpuEnabledGet( )**

vxCpuEnabledGet( )

NAME **vxCpuEnabledGet( )** – get a set of running CPUs

SYNOPSIS

```
cpuset_t vxCpuEnabledGet (void)
```

DESCRIPTION

This routine returns the set of CPUs that are running in the VxWorks SMP system. This set is updated at run-time as CPUs are enabled by the bootstrap CPU but the number of CPUs in the set can never be larger than the number of CPUs configured in the system. That is, the number of CPUs in the set cannot exceed the value returned by **vxCpuConfiguredGet( )**.

The default behaviour of VxWorks SMP is to take all configured CPUs out of reset at boot time. However this behaviour can be modified to only enable additional CPUs at a later point in time. This routine can therefore be used to obtain a true representation of the enabled CPUs as opposed to the number of configured CPUs.

Calling this routine in the uniprocessor version of VxWorks always returns a set that shows CPU0 as being the only enabled CPU. The coding example below shows a test case that could be used to test the expected behaviour of this routine in a uniprocessor environment.

```
STATUS test (void)
{
cpuset_t uniprocessorCpuSet;

/* Get the set of enabled CPUs */
uniprocessorCpuSet = vxCpuEnabledGet();

/* CPU 0 is supposed to be enabled.  Check it! */
if (CPUSET_ISSET(uniprocessorCpuSet, 0))
   {
   /*
    * First part of the test passed.  Now check that no other CPUs
    * are in the set.
    */

   CPUSET_CLR(uniprocessorCpuSet, 0);
   if (CPUSET_ISZERO(uniprocessorCpuSet))
       {
       /* No other CPUs in the set.  Test passed. */
       return (OK);
       }
   }

/*
 * Test failed.  Either CPU 0 was not in the set or other CPUs
 * were in the set.
 */
return (ERROR);
}
```

This routine can be called from both task or interrupt level.

RETURNS

A set of CPUs that have been enabled.

ERRNO

N/A

SEE ALSO

vxCpuLib, **vxCpuConfiguredGet( )**, cpuset

vxCpuIndexGet( )

NAME **vxCpuIndexGet( )** – get the index of the calling CPU

SYNOPSIS

```
unsigned int vxCpuIndexGet (void)
```

DESCRIPTION This routine returns the index of the CPU on which the calling task or ISR is running. The index is a number between 0 and N-1, where N is the number of CPUs configured in the SMP system. N is also the figure returned by **vxCpuConfiguredGet( )**. Calling this routine in the uniprocessor version of VxWorks returns 0, always. The value returned by this routine can easily be used as an index into an array of per-CPU objects that would have previously been allocated with the help of the **vxCpuConfiguredGet( )** routine.

Since tasks can migrate from one CPU to another in an SMP system, no guarantees are provided that the index is valid by the time program execution returns to the caller of this routine. For example, if a scheduling event takes place immediately after this call returns, it is possible for the caller to be running on a different CPU than it was at the time of the call. It is the responsibility of the caller to prevent task migration to another CPU while the index is being used. This can be done using **taskCpuLock( )**. If this routine is called from interrupt context the caller is guaranteed the index will be valid until the ISR returns. This is because ISRs do not migrate from one CPU to the other while they are running.

The purpose of this routine is different than that of **sysProcNumGet( )**, which is used to uniquely identify a node in an asymmetric multiprocessing environment. Should a node running VxWorks SMP exist in such an environment, **sysProcNumGet( )** on that node would always return the same value regardless of the CPU on which the calling thread is running.

RETURNS The index of the CPU on which the calling thread executes

ERRNO N/A

SEE ALSO **vxCpuLib**, **vxCpuConfiguredGet( )**

vxCr0Get( )

NAME **vxCr0Get( )** – get a content of the Control Register 0 (x86)

SYNOPSIS

```
int vxCr0Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 0.

RETURNS a value of the Control Register 0

ERRNO Not Available

SEE ALSO **vxLib**

vxCr0Set( )

NAME **vxCr0Set( )** – set a value to the Control Register 0 (x86)

SYNOPSIS

```
void vxCr0Set
    (
    int value  /* CR0 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 0.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxCr2Get( )

NAME **vxCr2Get( )** – get a content of the Control Register 2 (x86)

SYNOPSIS

```
int vxCr2Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 2.

RETURNS a value of the Control Register 2

ERRNO Not Available

SEE ALSO **vxLib**

vxCr2Set( )

NAME **vxCr2Set( )** – set a value to the Control Register 2 (x86)

SYNOPSIS

```
void vxCr2Set
    (
    int value  /* CR2 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 2.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxCr3Get( )

NAME **vxCr3Get( )** – get a content of the Control Register 3 (x86)

SYNOPSIS

```
int vxCr3Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 3.

RETURNS a value of the Control Register 3

ERRNO Not Available

SEE ALSO **vxLib**

vxCr3Set( )

NAME **vxCr3Set( )** – set a value to the Control Register 3 (x86)

SYNOPSIS

```
void vxCr3Set
    (
    int value  /* CR3 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 3.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxCr4Get( )

NAME **vxCr4Get( )** – get a content of the Control Register 4 (x86)

SYNOPSIS

```
int vxCr4Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 4.

RETURNS a value of the Control Register 4

ERRNO Not Available

SEE ALSO **vxLib**

vxCr4Set( )

NAME **vxCr4Set( )** – set a value to the Control Register 4 (x86)

SYNOPSIS

```
void vxCr4Set
    (
    int value  /* CR4 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 4.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxDrGet( )

NAME **vxDrGet( )** – get a content of the Debug Register 0 to 7 (x86)

SYNOPSIS

```
void vxDrGet
    (
    int * pDr0,  /* DR0 */
    int * pDr1,  /* DR1 */
    int * pDr2,  /* DR2 */
    int * pDr3,  /* DR3 */
    int * pDr4,  /* DR4 */
    int * pDr5,  /* DR5 */
    int * pDr6,  /* DR6 */
    int * pDr7   /* DR7 */
    )
```

DESCRIPTION This routine gets a content of the Debug Register 0 to 7.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxDrSet( )

NAME **vxDrSet( )** – set a value to the Debug Register 0 to 7 (x86)

SYNOPSIS

```
void vxDrSet
    (
    int dr0,  /* DR0 */
    int dr1,  /* DR1 */
    int dr2,  /* DR2 */
    int dr3,  /* DR3 */
    int dr4,  /* DR4 */
    int dr5,  /* DR5 */
    int dr6,  /* DR6 */
    int dr7   /* DR7 */
    )
```

DESCRIPTION This routine sets a value to the Debug Register 0 to 7.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxEflagsGet( )

NAME **vxEflagsGet( )** – get a content of the EFLAGS register (x86)

SYNOPSIS

```
int vxEflagsGet (void)
```

DESCRIPTION This routine gets a content of the EFLAGS register

RETURNS a value of the EFLAGS register

ERRNO Not Available

SEE ALSO **vxLib**

vxEflagsSet( )

NAME **vxEflagsSet( )** – set a value to the EFLAGS register (x86)

SYNOPSIS

```
void vxEflagsSet
    (
    int value   /* EFLAGS value */
    )
```

DESCRIPTION This routine sets a value to the EFLAGS register

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxGdtrGet( )

NAME
vxGdtrGet( ) – get a content of the Global Descriptor Table Register (x86)

SYNOPSIS
```
void vxGdtrGet
    (
    long long int * pGdtr  /* memory to store GDTR */
    )
```

DESCRIPTION
This routine gets a content of the Global Descriptor Table Register

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
vxLib

vxIdtrGet( )

NAME
vxIdtrGet( ) – get a content of the Interrupt Descriptor Table Register (x86)

SYNOPSIS
```
void vxIdtrGet
    (
    long long int * pIdtr  /* memory to store IDTR */
    )
```

DESCRIPTION
This routine gets a content of the Interrupt Descriptor Table Register

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
vxLib

vxLdtrGet( )

NAME
vxLdtrGet( ) – get a content of the Local Descriptor Table Register (x86)

SYNOPSIS
```
void vxLdtrGet
```

```
    (
    long long int * pLdtr  /* memory to store LDTR */
    )
```

DESCRIPTION This routine gets a content of the Local Descriptor Table Register

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxMemArchProbe( )

NAME **vxMemArchProbe( )** – architecture specific part of vxMemProbe

SYNOPSIS

```
STATUS vxMemArchProbe
    (
    FAST char * adrs,     /* address to be probed           */
    int         mode,     /* VX_READ or VX_WRITE            */
    int         length,   /* 1, 2, 4, or 8                  */
    FAST char * pVal      /* where to return value,         */
                          /* or ptr to value to be written */
    )
```

DESCRIPTION This is the routine implementing the architecture specific part of the vxMemProbe routine. It traps the relevant exceptions while accessing the specified address. If an exception occurs, then the result will be **ERROR**. If no exception occurs then the result will be **OK**.

RETURNS **OK** or **ERROR** if an exception occurred during access.

ERRNO Not Available

SEE ALSO **vxLib**

vxMemProbe( )

NAME **vxMemProbe( )** – probe an address for a bus error

SYNOPSIS

```
STATUS vxMemProbe
```

```
(
FAST char * adrs,    /* address to be probed           */
int         mode,    /* VX_READ or VX_WRITE            */
int         length,  /* 1, 2, 4, or 8                  */
FAST char * pVal     /* where to return value,         */
                     /* or ptr to value to be written */
)
```

DESCRIPTION

This routine probes a specified address to see if it is readable or writable, as specified by *mode*. The address is read or written as 1, 2, or 4 bytes, as specified by *length* (values other than 1, 2, or 4 yield unpredictable results). If the probe is a **VX_READ** (0), the value read is copied to the location pointed to by *pVal*. If the probe is a **VX_WRITE** (1), the value written is taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of 1, 2, or 4 bytes, as specified by *length*.

Note that only bus errors are trapped during the probe, and that the access must otherwise be valid (i.e., it must not generate an address error).

EXAMPLE

```
testMem (adrs)
    char *adrs;
    {
    char testW = 1;
    char testR;

    if (vxMemProbe (adrs, VX_WRITE, 1, &testW) == OK)
        printf ("value %d written to adrs %x\en", testW, adrs);

    if (vxMemProbe (adrs, VX_READ, 1, &testR) == OK)
        printf ("value %d read from adrs %x\en", testR, adrs);
    }
```

MODIFICATION

The BSP can modify the behaviour of **vxMemProbe( )** by supplying an alternate routine and placing the address in the global variable _func_vxMemProbeHook. The BSP routine will be called instead of the architecture specific routine **vxMemArchProbe( )**.

RETURNS

OK, or **ERROR** if the probe caused a bus error or was misaligned.

ERRNO

Not Available

SEE ALSO

vxLib, **vxMemArchProbe( )**

vxMemProbe( )

NAME

vxMemProbe( ) – probe an address for a bus error

SYNOPSIS

```
STATUS vxMemProbe
```

```
    (
    FAST char * pAdrs,    /* address to be probed */
    int         mode,     /* VX_READ or VX_WRITE */
    int         length,   /* 1, 2, or 4 */
    char *      pVal      /* Data source if VX_WRITE; destination if VX_READ
*/
    )
```

DESCRIPTION

This routine probes a specified address to see if it is readable or writable, as specified by *mode*. The address will be read or written according to the requested *length*. The provided pointer must be naturally aligned to the requested *length*.

If the requested mode is **VX_READ**, the value read will be copied to the location pointed to by *pVal*. If the requested mode is **VX_WRITE**, the value written will be taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of length *length*.

Note that only data bus errors (machine check exception, data access exception) are trapped during the probe, and that the access must be otherwise valid (i.e., not generate an address error).

EXAMPLE

```
testMem (adrs)
   char *adrs;
   {
   char testW = 1;
   char testR;

   if (vxMemProbe (adrs, VX_WRITE, 1, &testW) == OK)
       printf ("value %d written to adrs %x\en", testW, adrs);

   if (vxMemProbe (adrs, VX_READ, 1, &testR) == OK)
       printf ("value %d read from adrs %x\en", testR, adrs);
   }
```

MODIFICATION

The BSP can modify the behaviour of this routine by supplying an alternate routine and placing the address of the routine in the global variable _func_vxMemProbeHook. The BSP routine will be called instead of the architecture specific routine **vxMemArchProbe( )**.

RETURNS

OK if the probe is successful, or
ERROR if the probe caused a bus error.

ERRNO

Not Available

SEE ALSO

vxMemProbeLib, **vxMemArchProbe( )**

vxMemProbeInit( )

NAME
vxMemProbeInit( ) – add vxMemProbeTrap exception handler to exc handler chain

SYNOPSIS
```
STATUS vxMemProbeInit (void)
```

DESCRIPTION
Add the vxMemProbe exception handler hook to the exception handler chain called by excExcHandle

RETURNS
OK if initialization **OK** else **ERROR**

ERRNO
Not Available

SEE ALSO
vxMemProbeLib

vxPowerDown( )

NAME
vxPowerDown( ) – place the processor in reduced-power mode (PowerPC, SH)

SYNOPSIS
```
UINT32 vxPowerDown (void)
```

DESCRIPTION
This routine activates the reduced-power mode if power management is enabled. It is called by the scheduler when the kernel enters the idle loop. The power management mode is selected by **vxPowerModeSet( )**.

RETURNS
OK, or **ERROR** if power management is not supported or if external interrupts are disabled.

ERRNO
Not Available

SEE ALSO
vxLib, **vxPowerModeSet( )**, **vxPowerModeGet( )**

vxPowerModeGet( )

NAME
vxPowerModeGet( ) – get the power management mode (PowerPC, SH, x86)

SYNOPSIS
```
UINT32 vxPowerModeGet (void)
```

DESCRIPTION
This routine returns the power management mode set by **vxPowerModeSet( )**.

RETURNS The power management mode, or **ERROR** if no mode has been selected or if power management is not supported.

ERRNO Not Available

SEE ALSO **vxLib**, **vxPowerModeSet( )**, **vxPowerDown( )**

vxPowerModeSet( )

NAME **vxPowerModeSet( )** – set the power management mode (PowerPC, SH, x86)

SYNOPSIS

```
STATUS vxPowerModeSet
    (
    UINT32 mode  /* power management mode to select */
    )
```

DESCRIPTION This routine selects the power management mode to be activated when **vxPowerDown( )** is called. **vxPowerModeSet( )** is normally called in the BSP initialization routine **sysHwInit( )**.

USAGE PPC Power management modes include the following:

VX_POWER_MODE_DISABLE (0x1)
Power management is disabled; this prevents the MSR(POW) bit from being set (all PPC).

VX_POWER_MODE_FULL (0x2)
All CPU units are active while the kernel is idle (PPC603, PPCEC603 and PPC860 only).

VX_POWER_MODE_DOZE (0x4)
Only the decrementer, data cache, and bus snooping are active while the kernel is idle (PPC603, PPCEC603 and PPC860).

VX_POWER_MODE_NAP (0x8)
Only the decrementer is active while the kernel is idle (PPC603, PPCEC603 and PPC604).

VX_POWER_MODE_SLEEP (0x10)
All CPU units are inactive while the kernel is idle (PPC603, PPCEC603 and PPC860 - not recommended for the PPC603 and PPCEC603 architecture).

VX_POWER_MODE_DEEP_SLEEP (0x20)
All CPU units are inactive while the kernel is idle (PPC860 only - not recommended).

VX_POWER_MODE_DPM (0x40)
Dynamic Power Management Mode (PPC603 and PPCEC603 only).

VX_POWER_MODE_DOWN (0x80)
Only a hard reset causes an exit from power-down low power mode (PPC860 only - not recommended).

USAGE SH

Power management modes include the following:

VX_POWER_MODE_DISABLE (0x0)
Power management is disabled.

VX_POWER_MODE_SLEEP (0x1)
The core CPU is halted, on-chip peripherals operating, external memory refreshing.

VX_POWER_MODE_DEEP_SLEEP (0x2)
The core CPU is halted, on-chip peripherals operating, external memory self-refreshing (SH-4 only).

VX_POWER_MODE_USER (0xff)
Set up to three 8-bit standby registers with user-specified values:

```
vxPowerModeSet (VX_POWER_MODE_USER | sbr1<<8 | sbr2<<16 | sbr3<<24);
```

The sbr1 value is written to the STBCR or SBYCR1, sbr2 is written to the STBCR2 or SBYCR2, and sbr3 is written to the STBCR3 register (when available), depending on the SH processor type.

USAGE X86

vxPowerModeSet( ) is called in the BSP initialization routine **sysHwInit( )**. Power management modes include the following:

VX_POWER_MODE_DISABLE (0x1)
Power management is disable: this prevents halting the CPU.

VX_POWER_MODE_AUTOHALT (0x4)
Power management is enable: this allows halting the CPU.

RETURNS

OK, or **ERROR** if *mode* is incorrect or not supported by the processor.

ERRNO

Not Available

SEE ALSO

vxLib, **vxPowerModeGet( )**, **vxPowerDown( )**

vxSSDisable( )

NAME

vxSSDisable( ) – disable the superscalar dispatch (MC68060)

SYNOPSIS

```
void vxSSDisable (void)
```

DESCRIPTION This function resets the ESS bit of the Processor Configuration Register (PCR) to disable the superscalar dispatch.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxSSEnable( )

NAME **vxSSEnable( )** – enable the superscalar dispatch (MC68060)

SYNOPSIS

```
void vxSSEnable (void)
```

DESCRIPTION This function sets the ESS bit of the Processor Configuration Register (PCR) to enable the superscalar dispatch.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxTas( )

NAME **vxTas( )** – C-callable atomic test-and-set primitive

SYNOPSIS

```
BOOL vxTas
    (
    void * address   /* address to test and set */
    )
```

DESCRIPTION This routine provides a C-callable interface to a test-and-set instruction. The test-and-set instruction is executed on the specified address. The architecture test-and-set instruction is:

68K	**tas**
x86	**lock bts**
SH	**tas.b**
ARM	**swpb**

This routine is equivalent to **sysBusTas( )** in **sysLib**.

MIPS Because VxWorks does not support the MIPS MMU, only kseg0 and kseg1 addresses are accepted; other addresses return **FALSE**.

NOTE X86 BTS "Bit Test and Set" instruction is executed with LOCK instruction prefix to lock the Bus during the execution. The bit position 0 is toggled.

NOTE SH The SH version of **vxTas( )** simply executes the **tas.b** instruction, and the test-and-set (atomic read-modify-write) operation may require an external bus locking mechanism on some hardware. In this case, wrap the **vxTas( )** with a bus locking and unlocking code in the **sysBusTas( )**.

RETURNS **TRUE** if the value had not been set (but is now), or **FALSE** if the value was set already.

ERRNO Not Available

SEE ALSO **vxLib**, **sysBusTas( )**

vxTssGet( )

NAME **vxTssGet( )** – get a content of the TASK register (x86)

SYNOPSIS

```
int vxTssGet (void)
```

DESCRIPTION This routine gets a content of the TASK register

RETURNS a value of the TASK register

ERRNO Not Available

SEE ALSO **vxLib**

vxTssSet( )

NAME **vxTssSet( )** – set a value to the TASK register (x86)

SYNOPSIS

```
void vxTssSet
```

```
    (
    int value  /* TASK register value */
    )
```

DESCRIPTION This routine sets a value to the TASK register

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxLib**

vxbFileNvRamGet( )

NAME **vxbFileNvRamGet( )** – get the contents of non-volatile RAM

SYNOPSIS

```
STATUS vxbFileNvRamGet
    (
    char * fileName,  /* name of NVRam file */
    char   *string,   /* where to copy non-volatile RAM    */
    int    strLen,    /* maximum number of bytes to copy   */
    int    offset     /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION This routine copies the contents of non-volatile memory into a specified string. The string is terminated with an EOS.

RETURNS **OK**, or **ERROR** if parameters are invalid, the file cannot be opened, or cannot read from the file

ERRNO Not Available

SEE ALSO **vxbFileNvRam**, **vxbFileNvRamSet( )**

vxbFileNvRamRegister( )

NAME

vxbFileNvRamRegister( ) – register vxbFileNvRam driver

SYNOPSIS

```
void vxbFileNvRamRegister(void)
```

DESCRIPTION

This routine registers the vxbFileNvRam driver and device recognition data with the vxBus subsystem.

RETURNS

none

ERRNO

Not Available

SEE ALSO

vxbFileNvRam

vxbFileNvRamSet( )

NAME

vxbFileNvRamSet( ) – write to non-volatile RAM

SYNOPSIS

```
STATUS vxbFileNvRamSet
    (
    char *fileName,  /* name of NVRam file */
    char *string,    /* string to be copied into non-volatile RAM */
    int  strLen,     /* maximum number of bytes to copy           */
    int  offset      /* byte offset into non-volatile RAM         */
    )
```

DESCRIPTION

This routine copies a specified string into non-volatile RAM.

RETURNS

OK, or **ERROR** if parameters are invalid, cannot open the
nvram file, or cannot write to the nvram file.

ERRNO

Not Available

SEE ALSO

vxbFileNvRam, **vxbFileNvRamGet( )**

vxbFileNvRampDrvCtrlShow( )

NAME **vxbFileNvRampDrvCtrlShow( )** – show pDrvCtrl for template controller

SYNOPSIS

```
int vxbFileNvRampDrvCtrlShow
    (
    VXB_DEVICE_ID pInst,
    int           verboseLevel
    )
```

DESCRIPTION This routine prints information about the instance to to system console. This is not integrated with **vxBusShow**.

RETURNS: 0, always

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbFileNvRam**

vxbIntelIchStorageRegister( )

NAME **vxbIntelIchStorageRegister( )** – register driver with vxbus

SYNOPSIS

```
void vxbIntelIchStorageRegister (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO

SEE ALSO **vxbIntelIchStorage**

vxbNonVolGet( )

NAME

vxbNonVolGet( ) – get the contents of non-volatile RAM

SYNOPSIS

```
STATUS vxbNonVolGet
    (
    char * drvName,  /* requestor's name */
    int    drvUnit,  /* requestor's unit number */
    char * buff,     /* buffer to copy non-volatile RAM into */
    int    offset,   /* offset from start of allocation unit */
    int    strLen    /* maximum number of bytes to copy */
    )
```

DESCRIPTION

This routine reads information from a non-volatile memory device and stores it in the caller-provided buffer.

The caller identifies itself by name and unit number. This routine is typically called by a device driver. In this case, the specified name is the name of the device driver, and the unit number is the unit number of the device/driver instance. However, other modules which use NVRam may make use of this routine as well, such as BOOTLINE. In this case, unit number should be set to zero or a number specific to the module.

The amount of data copied is the size specified. If the specified size is greater than the size of the NVram segment allocated to the device, the behavior is undefined.

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

vxbNonVolLib, **vxbNonVolSet( )**

vxbNonVolLibInit( )

NAME

vxbNonVolLibInit( ) – Non Volatile RAM library initialization

SYNOPSIS

```
void vxbNonVolLibInit(void)
```

DESCRIPTION

none

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO **vxbNonVolLib**

vxbNonVolSet( )

NAME **vxbNonVolSet( )** – write to non-volatile memory

SYNOPSIS

```
STATUS vxbNonVolSet
    (
    char * drvName,  /* requestor's name */
    int    drvUnit,  /* requestor's unit number */
    char * buff,     /* buffer to copy from, into non-volatile RAM */
    int    offset,   /* offset from start of allocation unit */
    int    strLen    /* maximum number of bytes to copy */
    )
```

DESCRIPTION This routine reads information from the caller-provided buffer and stores it in a non-volatile memory device.

The caller identifies itself by name and unit number. This routine is typically called by a device driver. In this case, the specified name is the name of the device driver, and the unit number is the unit number of the device/driver instance. However, other modules which use NVRam may make use of this routine as well, such as BOOTLINE. In this case, unit number should be set to zero or a number specific to the module.

The amount of data copied is the size specified. If the specified size is greater than the size of the NVram segment allocated to the device, the behavior is undefined.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbNonVolLib**, **vxbNonVolGet( )**

vxbSI31xxStorageRegister( )

NAME **vxbSI31xxStorageRegister( )** – register driver with vxbus

SYNOPSIS

```
void vxbSI31xxStorageRegister (void)
```

DESCRIPTION none

RETURNS N/A

ERRNO

SEE ALSO **vxbSI31xxStorage**

vxsimHostCpuVarsInit( )

NAME **vxsimHostCpuVarsInit( )** – intialize per cpu variable pointers

SYNOPSIS

```
void vxsimHostCpuVarsInit (void)
```

DESCRIPTION This routine initializes VxWorks pointer to addresses in host binary in orde address is the same on every cpu but the value can be different.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxsimHostArchLib**

vxsimHostDllLoad( )

NAME **vxsimHostDllLoad( )** – load the given Dll to VxSim.

SYNOPSIS

```
STATUS vxsimHostDllLoad
    (
    char * dllName  /* name of the DLL to load      */
    )
```

DESCRIPTION This routine loads the given Dll to VxSim. The Dll is loaded from current directory, using an absolute path name, from the System Dll search path, or from VxSim Dll path *WIND_BASE*/host/*HOST_TYPE*/lib/vxsim.

SMP CONSIDERATIONS

dllName parameter must be in kernel memory. DLL is only loaded on CPU0.

RETURNS **OK**, or **ERROR** if load failed.

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostMmuCurrentSet( )

NAME **vxsimHostMmuCurrentSet( )** – set current translation table mapping

SYNOPSIS

```
STATUS vxsimHostMmuCurrentSet
    (
    MMU_TRANS_TBL * pTransTbl  /* translation table to set */
    )
```

DESCRIPTION Set mapping corresponding to specified translation table. This routine only affects the current CPU. Vxsim host binary makes sure the translation table can not be updated while beeing read.

RETURNS **OK** always

ERRNO Not Available

SEE ALSO **vxsimHostArchLib**

vxsimHostMmuProtect( )

NAME **vxsimHostMmuProtect( )** – set/clear protection on mmu pages

SYNOPSIS

```
STATUS vxsimHostMmuProtect
    (
    MMU_TRANS_TBL * pTransTbl,  /* translation table */
    VIRT_ADDR       addr,       /* address to check */
    UINT32          state,      /* protection state */
    UINT32          numPages    /* number of pages */
    )
```

DESCRIPTION This routine sets or clear protection flags on mmu pages. It does not updtae corresponding PTE.

RETURNS **OK** or **ERROR**

ERRNO Not Available

SEE ALSO **vxsimHostArchLib**

vxsimHostProcAddrGet( )

NAME **vxsimHostProcAddrGet( )** – return the address of a host API

SYNOPSIS

```
FUNCPTR vxsimHostProcAddrGet
    (
    char * routineName  /* host API  name          */
    )
```

DESCRIPTION This routine returns the address of a host API which name is given in parameter. No error message are displayed if the host API is not found.

SMP CONSIDERATIONS

routineName parameter must be in kernel memory.

RETURNS The address of the routine, or **NULL** if not found.

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostProcCall( )

NAME **vxsimHostProcCall( )** – call a host routine

SYNOPSIS

```
UINT32 vxsimHostProcCall
    (
    FUNCPTR rtnAddr,  /* routine to be called */
    UINT32  arg0,     /* routine arguments */
    UINT32  arg1,
    UINT32  arg2,
    UINT32  arg3,
    UINT32  arg4,
    UINT32  arg5,
    UINT32  arg6,
    UINT32  arg7,
    UINT32  arg8
    )
```

DESCRIPTION
This routine calls a host routine whose address was previously retrieved through **vxsimHostProcAddrGet( )**. On a SMP system it is the only way to use safely the value returned by **vxsimHostProcAddrGet( )** as this value is only guaranteed to be correct on CPUi that performed **vxsimHostProcAddrGet( )**.

RETURNS
routine return value as an UINT32

ERRNO
Not Available

SEE ALSO
vxsimHostArchLib

vxsimHostSioBaudRateSet( )

NAME
vxsimHostSioBaudRateSet( ) – set SIO device transfert rate

SYNOPSIS
```
STATUS vxsimHostSioBaudRateSet
    (
    SIO_ID sioId,
    int    baudRate
    )
```

DESCRIPTION
This routine sets SIO device transfert rate

RETURNS
OK or **ERROR**

ERRNO
N/A

SEE ALSO
vxsimHostArchLib

vxsimHostSioClose( )

NAME
vxsimHostSioClose( ) – close SIO device

SYNOPSIS
```
STATUS vxsimHostSioClose
    (
    SIO_ID sioId  /* sio descriptor */
    )
```

DESCRIPTION This routine closes specified SIO device.

RETURNS number of bytes read

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostSioIntVecGet( )

NAME **vxsimHostSioIntVecGet( )** – get SIO device interrupt vector

SYNOPSIS

```
int vxsimHostSioIntVecGet
    (
    SIO_ID sioId  /* sio descriptor */
    )
```

DESCRIPTION This routine gets interrupt vector associated with a SIO device.

RETURNS interrupt vector

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostSioModeSet( )

NAME **vxsimHostSioModeSet( )** – set SIO device mode (poll/interrupt)

SYNOPSIS

```
STATUS vxsimHostSioModeSet
    (
    SIO_ID sioId,  /* sio descriptor */
    int    mode    /* 0 = poll, 1 = interrupt */
    )
```

DESCRIPTION This routine sets SIO device interrupt mode.

RETURNS **OK** or **ERROR**

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostSioOpen( )

NAME **vxsimHostSioOpen( )** – open SIO device

SYNOPSIS

```
STATUS vxsimHostSioOpen
    (
    SIO_ID sioId  /* sio descriptor */
    )
```

DESCRIPTION This routine opens specified SIO device.

RETURNS **OK** or **ERROR**

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostSioRead( )

NAME **vxsimHostSioRead( )** – read SIO device into buffer

SYNOPSIS

```
int vxsimHostSioRead
    (
    SIO_ID sioId,
    char * buf,
    int    len
    )
```

DESCRIPTION This routine reads specified SIO device.

RETURNS number of bytes read

ERRNO N/A

SEE ALSO **vxsimHostArchLib**

vxsimHostSioWrite( )

NAME
vxsimHostSioWrite( ) – write buffer to SIO device

SYNOPSIS
```
int vxsimHostSioWrite
    (
    SIO_ID sioId,
    char * buf,
    int    len
    )
```

DESCRIPTION
This routine writes specified buffer to specified SIO device.

RETURNS
number of bytes written

ERRNO
N/A

SEE ALSO
vxsimHostArchLib

w( )

NAME
w( ) – print a summary of each task's pending information, task by task

SYNOPSIS
```
void w
    (
    int taskNameOrId  /* task name or task ID */
    )
```

DESCRIPTION
This routine shows a summary of each task's pending information, if *taskNameOrId* is equal to 0. Otherwise, it shows a summary for the specified task.

This routine doesn't support POSIX semaphores and message queues. This command doesn't support pending signals.

List of object types that are recognized:

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usrLib, **tw( )**, the VxWorks programmer guides.

wdCancel( )

NAME

wdCancel( ) – cancel a currently counting watchdog

SYNOPSIS

```
STATUS wdCancel
    (
    WDOG_ID wdId  /* ID of watchdog to cancel */
    )
```

DESCRIPTION

This routine cancels a currently running watchdog timer by zeroing its delay count. Watchdog timers may be canceled from interrupt level.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK, or **ERROR** if the watchdog timer cannot be canceled.

ERRNO

Not Available

SEE ALSO

wdLib, **wdStart( )**

wdCreate( )

NAME

wdCreate( ) – create a watchdog timer

SYNOPSIS

```
WDOG_ID wdCreate (void)
```

DESCRIPTION

This routine creates a watchdog timer by allocating a WDOG structure in memory.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

The ID for the watchdog created, or **NULL** if memory is insufficient.

ERRNO

Not Available

SEE ALSO

wdLib, **wdDelete( )**

wdDelete( )

NAME

wdDelete( ) – delete a watchdog timer

SYNOPSIS

```
STATUS wdDelete
    (
    WDOG_ID wdId  /* ID of watchdog to delete */
    )
```

DESCRIPTION

This routine de-allocates a watchdog timer. The watchdog will be removed from the timer queue if it has been started. This routine complements **wdCreate( )**.

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS

OK, or **ERROR** if the watchdog timer cannot be de-allocated.

ERRNO

Not Available

SEE ALSO

wdLib, **wdCreate( )**

wdInitialize( )

NAME

wdInitialize( ) – initialize a pre-allocated watchdog.

SYNOPSIS

```
WDOG_ID wdInitialize
    (
    char * pWdMem
    )
```

DESCRIPTION

This routine initializes a watchdog that has been pre-allocated (i.e. by the **VX_WDOG** macro).

The following example illustrates use of the **VX_WDOG** macro and this function together to instantiate a watchdog statically (without using any dynamic memory allocation):

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

VX_WDOG(myWdog);     /* declare the watchdog */
WDOG_ID myWdogId;    /* watchdog ID for further operations */

STATUS initializeFunction (void)
    {
    if ((myWdogId = wdInitialize (myWdog)) == NULL)
```

```
          return (ERROR);       /* initialization failed */
    else
          return (OK);
    }
```

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS The watchdog ID, or **NULL** on error.

ERRNO N/A

SEE ALSO **wdLib**

---

wdShow( )

NAME **wdShow( )** – show information about a watchdog

SYNOPSIS

```
STATUS wdShow
    (
    WDOG_ID wdId  /* watchdog to display */
    )
```

DESCRIPTION This routine displays the state of a watchdog.

EXAMPLE A summary of the state of a watchdog is displayed as follows:

```
    -> wdShow myWdId
    Watchdog Id          : 0x3dd46c
    State                : OUT_OF_Q
    Ticks Remaining      : 0
    Routine              : 0
    Parameter            : 0
```

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK** or **ERROR**.

ERRNO Not Available

SEE ALSO **wdShow**, **windsh**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

wdShowInit( )

NAME **wdShowInit( )** – initialize the watchdog show facility

SYNOPSIS

```
void wdShowInit (void)
```

DESCRIPTION This routine links the watchdog show facility into the VxWorks system. It is called automatically when the watchdog 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**.

RETURNS N/A

ERRNO Not Available

SEE ALSO **wdShow**

wdStart( )

NAME **wdStart( )** – start a watchdog timer

SYNOPSIS

```
STATUS wdStart
    (
    WDOG_ID wdId,       /* watchdog ID */
    int     delay,      /* delay count, in ticks */
    FUNCPTR pRoutine,   /* routine to call on time-out */
    int     parameter   /* parameter with which to call routine */
    )
```

DESCRIPTION This routine adds a watchdog timer to the system tick queue. The specified watchdog routine will be called from interrupt level after the specified number of ticks has elapsed. Watchdog timers may be started from interrupt level.

To replace either the timeout *delay* or the routine to be executed, call **wdStart( )** again with the same *wdId*; only the most recent **wdStart( )** on a given watchdog ID has any effect. (If your application requires multiple watchdog routines, use **wdCreate( )** to generate separate a watchdog ID for each.) To cancel a watchdog timer before the specified tick count is reached, call **wdCancel( )**.

Watchdog timers execute only once, but some applications require periodically executing timers. To achieve this effect, the timer routine itself must call **wdStart( )** to restart the timer on each invocation.

WARNING The watchdog routine runs in the context of the system-clock ISR; thus, it is subject to all ISR restrictions.

NOTE watchdog routine invocation can be deferred. As such isrIdCurrent is either a valid **ISR_ID** or is **NULL** in the case of deferral

SMP CONSIDERATIONS

This API is spinlock and intCpuLock restricted.

RETURNS **OK**, or **ERROR** if the watchdog timer cannot be started.

ERRNO Not Available

SEE ALSO **wdLib**, **wdCancel( )**

wdbMdlSymSyncLibInit( )

NAME **wdbMdlSymSyncLibInit( )** – initialize modules and symbols synchronization library

SYNOPSIS

```
void wdbMdlSymSyncLibInit (void)
```

DESCRIPTION This routine initializes the stuff needed by the modules and symbols synchronization.

RETURNS N/A

ERRNO N/A

SEE ALSO **wdbMdlSymSyncLib**

wdbSystemSuspend( )

NAME **wdbSystemSuspend( )** – suspend the system

SYNOPSIS

```
STATUS wdbSystemSuspend (void)
```

DESCRIPTION
This routine transfers control from the run time system to the WDB agent running in external mode. In order to give back the control to the system it must be resumed by the external WDB agent.

EXAMPLE
The code below, called in a vxWorks application, suspends the system :

```
if (wdbSystemSuspend () != OK)
    printf ("External mode is not supported by the WDB agent.\n");
```

From a host tool, we can detect that the system is suspended.

First, attach to the target server :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
```

Then, you can get the agent mode :

```
wtxtcl> wtxAgentModeGet
AGENT_MODE_EXTERN
```

To get the status of the system context, execute :

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_SUSPENDED
```

In order to resume the system, simply execute :

```
wtxtcl>  wtxContextResume CONTEXT_SYSTEM 0
0
```

You will see that the system is now running :

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_RUNNING
```

RETURNS
OK upon successful completion, **ERROR** if external mode is not supported by the WDB agent.

ERRNO
Not Available

SEE ALSO
wdbLib

wdbUserEvtLibInit( )

NAME
wdbUserEvtLibInit( ) – include the WDB user event library

SYNOPSIS

```
void wdbUserEvtLibInit (void)
```

DESCRIPTION This null routine is provided so that **wdbUserEvtLib** can be linked into the system. If the component **INCLUDE_WDB_USER_EVENT** is included at configuration time, **wdbUserEvtLibInit( )** is automatically called.

RETURNS N/A

ERRNO Not Available

SEE ALSO **wdbUserEvtLib**

wdbUserEvtPost( )

NAME **wdbUserEvtPost( )** – post a user event string to host tools

SYNOPSIS

```
STATUS wdbUserEvtPost
    (
    char * event  /* event string to send */
    )
```

DESCRIPTION This routine posts the string *event* to host tools that have registered for it. Host tools will receive a USER WTX event string. The maximum size of the event is **WDB_MAX_USER_EVT_SIZE** (defined in $**WIND_BASE/target/h/wdb/wdbLib.h**).

EXAMPLE The code below sends a WDB user event to host tools :

```
char * message = "Alarm: reactor overheating !!!";

if (wdbUserEvtPost (message) != OK)
    printf ("Can't send alarm message to host tools");
```

This event will be received by host tools that have registered for it. For example a WTX TCL based tool would do :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
wtxtcl> wtxRegisterForEvent "USER.*"
0
wtxtcl> wtxEventGet
USER Alarm: reactor overheating !!!
```

Host tools can register for more specific user events :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
wtxtcl> wtxRegisterForEvent "USER Alarm.*"
0
wtxtcl> wtxEventGet
USER Alarm: reactor overheating !!!
```

In this piece of code, only the USER events beginning with "Alarm" will be received.

RETURNS **OK** upon successful completion or **ERROR** if unable to send the event to the host or if the size of the event is greater than **WDB_MAX_USER_EVT_SIZE**.

ERRNO Not Available

SEE ALSO **wdbUserEvtLib**

wim( )

NAME **wim( )** – return the contents of the window invalid mask register (SimSolaris)

SYNOPSIS

```
int wim
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the window invalid mask register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS The contents of the window invalid mask register.

ERRNO Not Available

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

windPwrDownRtnSet( )

NAME **windPwrDownRtnSet( )** – register a BSP power-down function

SYNOPSIS

```
void windPwrDownRtnSet
    (
    WIND_PWR_DOWN_RTN dRtn  /* power down function pointer from BSP */
    )
```

WARNING This routine is deprecated. Calling this function is a nop.

This routine registers a BSP power-down function with WIND CPU power management. The function registered will be called when the WIND kernel decides that the CPU can be powered off. Note that the power-down function will not be invoked while the CPU power mode is set to windPwrModeOff.

The power-down function is passed two parameters: a **WIND_PWR_MODE** power mode and a ULONG nTicks.

The power mode parameter is the current kernel power mode that is in effect (as set by **windPwrModeSet( )**) when the WIND kernel goes idle. nTicks is the maximum number of ticks that the WIND kernel is willing to sleep for before it must wake up and perform some work, such as scheduling a task. nTicks may be passed as **WAIT_FOREVER** (0L), which indicates that the kernel has requirements as to when it is woken up next.

The routine registered is invoked with interrupts locked and is not allowed to make any WIND kernel calls either directly or indirectly. If these must be made, the only option is for the power-down function to perform a windPwrModeSet (windPwrModeOff) and arrange for an interrupt (a software or hardware interrupt) to occur to make the WIND kernel calls on its behalf. Such an interrupt will occur after the WIND kernel unlocks interrupts which it does after invoking the registered power-down routine, aborting its subsequent call to **vxArchPowerDown( )** to run the interrupt and process any kernel work the interrupt makes.

SMP CONSIDERATIONS

In an SMP environment it is possible for some CPUs to be idle and others to be executing tasks or ISRs. The registered power-down routine is called when a CPU goes idle regardless of the state of other CPUs in the SMP system. The routine is executed by the CPU that is going idle. The routine must ensure it does not perform any power-down actions that would disrupt execution on non-idle CPUs.

RETURNS N/A

ERRNO N/A

SEE ALSO **windPwrLib**, **kernelIsCpuIdle( )**, **kernelIsSystemIdle( )**

windPwrModeGet( )

NAME **windPwrModeGet( )** – Get the current power mode

SYNOPSIS

```
WIND_PWR_MODE windPwrModeGet(void)
```

DESCRIPTION This routine is called whenever the BSP needs the current power mode.

RETURNS **WIND_PWR_MODE**

ERRNO
N/A

SEE ALSO
windPwrLib, **windPwrModeSet( )**

windPwrModeSet( )

NAME
windPwrModeSet( ) – Set the BSP power mode

SYNOPSIS
```
void windPwrModeSet
    (
    WIND_PWR_MODE mode  /* new power mode */
    )
```

DESCRIPTION
This routine is called during initialization and whenever the power mode is set by the BSP.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
windPwrLib, **windPwrModeGet( )**

windPwrUpRtnSet( )

NAME
windPwrUpRtnSet( ) – register a BSP power-up function

SYNOPSIS
```
void windPwrUpRtnSet
    (
    WIND_PWR_UP_RTN uRtn
    )
```

WARNING
This routine is deprecated. Calling this function is a nop.

This routine registers a BSP power-up function with WIND CPU power management. The function registered will be called whenever an interrupt exception occurs to wake up the CPU while it was powered off (or it was in the process of powering off).

The power-up function is passed two parameters: a **WIND_PWR_MODE** power mode and a pointer to a ULONG nTicks.

The power mode parameter is the current WIND kernel power mode that is in effect (as set by **windPwrModeSet( )**) when the kernel wakes up. The pointer to nTicks is to be set by the power-up function to inform the kernel how long it has slept, in ticks.

The routine registered is invoked with interrupts locked and is not allowed to make any WIND kernel calls either directly or indirectly. If these must be made, the only option is for the power-up function to arrange for an interrupt (a software or hardware interrupt) to occur to make the WIND kernel calls on its behalf. Such an interrupt will occur after the WIND kernel unlocks interrupts which it does after invoking the registered power-up routine. If the arranged interrupt is at a higher priority than the interrupt that is waking up the CPU, it will execute immediately, otherwise it will run afterwards.

SMP CONSIDERATIONS

In an SMP environment it is possible for some CPUs to be idle and others to be executing tasks or ISRs. The registered power-up routine is called when a CPU is awakened regardless of the state of other CPUs in the SMP system. The routine is executed by the CPU that is awakened. The routine must ensure it does not perform any power-up actions that would disrupt execution on other CPUs.

RETURNS N/A

ERRNO N/A

SEE ALSO **windPwrLib**, **kernelIsCpuIdle( )**, **kernelIsSystemIdle( )**

write( )

NAME **write( )** – write bytes to a file

SYNOPSIS

```
int write
    (
    int    fd,       /* file descriptor on which to write     */
    char   *buffer,  /* buffer containing bytes to be written */
    size_t nbytes    /* number of bytes to write              */
    )
```

DESCRIPTION This routine writes *nbytes* bytes from *buffer* to a specified file descriptor *fd*. It calls the device driver to do the work.

RETURNS The number of bytes written (if not equal to *nbytes*, an error has occurred), or **ERROR** if the file descriptor does not exist, the driver does not have a write routine, or the driver returns **ERROR**. If the driver does not have a write routine, errno is set to **ENOTSUP**.

ERRNO

EBADF
Bad file descriptor number.

ENOTSUP
Device driver does not support the write command.

ENXIO
Device and its driver are removed. **close( )** should be called to release this file descriptor.

Other
Other errors reported by device driver.

SEE ALSO

ioLib

wvAllObjsSet( )

NAME

wvAllObjsSet( ) – set instrumented state for all objects and classes

SYNOPSIS

```
void wvAllObjsSet
    (
    int mode  /* INSTRUMENT_ON or INSTRUMENT_OFF */
    )
```

DESCRIPTION

This routine enables or disables instrumentation for all object classes and instances in the system.

If *mode* is **INSTRUMENT_ON**, instrumentation is turned on; if it is **INSTRUMENT_OFF**, instrumentation is turned off. Any other value has no effect

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

RETURNS

N/A

ERRNO

SEE ALSO

wvLib, **wvSigInst( )**, **wvEventInst( )**, **wvObjInstAllClear( )**, **wvObjInst( )**

wvCurrentLogGet( )

NAME **wvCurrentLogGet( )** – return a pointer to the currently active System Viewer log

SYNOPSIS

```
WV_LOG * wvCurrentLogGet (void)
```

DESCRIPTION This routine returns a pointer to the currently active System Viewer log.

RETURNS Pointer to the log, or **NULL**

ERRNO

SEE ALSO **wvLib**, **wvCurrentLogSet( )**

wvCurrentLogListGet( )

NAME **wvCurrentLogListGet( )** – return a pointer to the System Viewer log list

SYNOPSIS

```
WV_LOG_LIST * wvCurrentLogListGet (void)
```

DESCRIPTION This routine returns a pointer to the System Viewer log list. It is not expected that there would be more than one list at any one time.

RETURNS Pointer to the log list, or **NULL**

ERRNO

SEE ALSO **wvLib**, **wvCurrentLogListSet( )**

wvCurrentLogListSet( )

NAME **wvCurrentLogListSet( )** – set the current log list

SYNOPSIS

```
void wvCurrentLogListSet
    (
    WV_LOG_LIST * pWvLogList
    )
```

DESCRIPTION
This routine selects a System Viewer log list for subsequent operations. It could be used after reboot, to choose a log in a persistent memory area.

RETURNS
N/A

ERRNO

SEE ALSO
wvLib, **wvCurrentLogListGet( )**

wvCurrentLogSet( )

NAME
wvCurrentLogSet( ) – select a System Viewer log as currently active

SYNOPSIS
```
void wvCurrentLogSet
    (
    WV_LOG * pWvLog
    )
```

DESCRIPTION
This routine selects a System Viewer log as the active one.

RETURNS
N/A

ERRNO

SEE ALSO
wvLib, **wvCurrentLogGet( )**

wvEdrInst( )

NAME
wvEdrInst( ) – instrument ED&R Events

SYNOPSIS
```
STATUS wvEdrInst
    (
    WV_INSTRUMENTATION_MODE mode  /* instrumentation mode */
    )
```

DESCRIPTION
This routine instruments ED&R Event activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for ED&R events is turned on; if *mode* is **INSTRUMENT_OFF**, instrumentation for ED&R Events is turned off. Any other value causes the current instrumentation state to be returned.

RETURNS **OK** or **ERROR**.

ERRNO

SEE ALSO **wvLib**

wvEvent( )

NAME **wvEvent( )** – log a user-defined event

SYNOPSIS

```
STATUS wvEvent
    (
    event_t usrEventId,  /* event */
    char *  buffer,      /* buffer */
    size_t  bufSize      /* buffer size */
    )
```

DESCRIPTION This routine logs a user event. Event logging must have been started with **wvEvtLogStart( )** or from the System Viewer GUI to use this routine. The *usrEventId* should be in the range 0-25535. A buffer of data can be associated with the event; *buffer* is a pointer to the start of the data block, and *bufSize* is its length in bytes.

SMP CONSIDERATIONS

This API is spinlock restricted.

RETURNS **OK**, or **ERROR** if the event can not be logged.

ERRNO

SEE ALSO **wvLib**, **dbgLib**, **e( )**

wvEventInst( )

NAME **wvEventInst( )** – instrument VxWorks Events

SYNOPSIS

```
int wvEventInst
    (
    WV_INSTRUMENTATION_MODE mode  /* INSTRUMENT_ON, INSTRUMENT_OFF */
    )
```

DESCRIPTION This routine instruments VxWorks Event activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for VxWorks events is turned on; if it is any other value (including **INSTRUMENT_OFF**), instrumentation for VxWorks Events is turned off.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

Parameters:

mode
: The required instrumentation mode. The value **INSTRUMENT_ON** enables instrumentation for events, **INSTRUMENT_OFF** disables it, and any other value causes the current state to be returned.

RETURNS The mode (**INSTRUMENT_ON** or **INSTRUMENT_OFF**) currently in force.

ERRNO

SEE ALSO **wvLib**

wvEvtClassClear( )

NAME **wvEvtClassClear( )** – clear the specified class of events from those being logged

SYNOPSIS

```
void wvEvtClassClear
    (
    UINT32 classDescription  /* description of evt classes to clear */
    )
```

DESCRIPTION This routine clears the class or classes described by *classDescription* from the set of classes currently being logged.

RETURNS N/A

ERRNO

SEE ALSO **wvLib**

wvEvtClassClearAll( )

NAME **wvEvtClassClearAll( )** – clear all classes of events from those logged

SYNOPSIS

```
void wvEvtClassClearAll (void)
```

DESCRIPTION This routine clears all classes of events so that no classes are logged if event logging is started.

RETURNS N/A

ERRNO

SEE ALSO **wvLib**

wvEvtClassGet( )

NAME **wvEvtClassGet( )** – get the current set of classes being logged

SYNOPSIS

```
UINT32 wvEvtClassGet (void)
```

DESCRIPTION This routine returns the set of classes currently being logged.

RETURNS The class description.

ERRNO

SEE ALSO **wvLib**

wvEvtClassSet( )

NAME **wvEvtClassSet( )** – set the class of events to log

SYNOPSIS

```
void wvEvtClassSet
    (
    UINT32 classDescription  /* description of evt classes to set */
    )
```

DESCRIPTION

This routine sets the class of events which are logged when event logging is started. *classDescription* can take the following values:

```
    WV_CLASS_1       /* Events causing context switches */
    WV_CLASS_2       /* Events causing task-state transitions */
    WV_CLASS_3       /* Events from object and system libraries */
```

See **wvLib** for more information about these classes, particularly Class 3.

RETURNS

N/A

ERRNO

SEE ALSO

wvLib, **wvObjInst( )**, **wvSigInst( )**, **wvEventInst( )**, **wvSalInst( )**.

wvEvtLogStart( )

NAME

wvEvtLogStart( ) – start logging events to the buffer

SYNOPSIS

```
STATUS wvEvtLogStart (void)
```

DESCRIPTION

This routine starts event logging. It also resets the timestamp mechanism so that it can be called more than once without stopping event logging.

RETURNS

OK, or **ERROR** if no buffer in use

ERRNO

SEE ALSO

wvLib

wvEvtLogStop( )

NAME

wvEvtLogStop( ) – stop logging events to the buffer

SYNOPSIS

```
void wvEvtLogStop (void)
```

DESCRIPTION

This routine turns off all event logging, including event-logging of objects and signals specifically requested by the user. In addition, it disables the timestamp facility.

RETURNS

N/A

ERRNO

SEE ALSO **wvLib**

wvFileUploadPathCreate( )

NAME **wvFileUploadPathCreate( )** – create a file for depositing event data

SYNOPSIS

```
UPLOAD_ID wvFileUploadPathCreate
    (
    char *fname,    /* name of file to create */
    int  openFlags  /* O_CREAT, O_TRUNC */
    )
```

DESCRIPTION This routine opens and initializes a file to receive uploaded events. The *openFlags* argument is passed on as the flags argument to the actual open call so that the caller can specify things like **O_TRUNC** and **O_CREAT**. The file is always opened as **O_WRONLY**, regardless of the value of *openFlags*.

RETURNS The **UPLOAD_ID**, or **NULL** if the file can not be opened or memory for the ID is not available.

ERRNO Not Available

SEE ALSO **wvFileUploadPathLib**, **wvFileUploadPathClose( )**

wvFileUploadPathLibInit( )

NAME **wvFileUploadPathLibInit( )** – initialize the **wvFileUploadPathLib** library

SYNOPSIS

```
STATUS wvFileUploadPathLibInit (void)
```

DESCRIPTION This routine initializes the library by pulling in the routines in this file for use with WindView. It is called during system configuration from **usrWindview.c**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **wvFileUploadPathLib**

wvFileUploadPathWrite( )

NAME **wvFileUploadPathWrite( )** – write to the event-destination file

SYNOPSIS

```
int wvFileUploadPathWrite
    (
    UPLOAD_ID pathId,  /* generic upload-path descriptor */
    char *    pStart,  /* address of data to write */
    size_t    size     /* number of bytes of data at pStart */
    )
```

DESCRIPTION This routine writes *size* bytes of data beginning at *pStart* to the file indicated by *pathId*.

RETURNS The number of bytes written, or **ERROR**.

ERRNO Not Available

SEE ALSO **wvFileUploadPathLib**

wvLibInit( )

NAME **wvLibInit( )** – initialize **wvLib** - first step

SYNOPSIS

```
void wvLibInit (void)
```

DESCRIPTION This routine starts initializing **wvLib**. Its actions should be performed before object creation, so it is called from **usrKernelInit( )** in **usrKernel.c**.

RETURNS N/A

ERRNO

SEE ALSO **wvLib**

wvLibInit2( )

NAME **wvLibInit2( )** – initialize **wvLib** - final step

SYNOPSIS

```
void wvLibInit2 (void)
```

DESCRIPTION This routine is called after **wvLibInit( )** to complete the initialization of **wvLib**. It should be called before starting any event logging.

RETURNS N/A

ERRNO

SEE ALSO **wvLib**

wvLogCountGet( )

NAME **wvLogCountGet( )** – return the number of logs in the curent log list

SYNOPSIS

```
int wvLogCountGet (void)
```

DESCRIPTION This routine returns the number of System Viewer logs in the current log list.

RETURNS number of logs

ERRNO

SEE ALSO **wvLib**, **wvLogFirstGet( )**, **wvLogNextGet( )**

wvLogCreate( )

NAME **wvLogCreate( )** – Create a System Viewer log

SYNOPSIS

```
WV_LOG * wvLogCreate
    (
    BUFFER_ID evtBuffer  /* event buffer to use */
    )
```

DESCRIPTION This routine creates a System Viewer log, and then inserts it into the System Viewer log list. If the routine encounters an error, then the insertion is not done, a node is not created, and the caller should delete the event buffer passed in.

RETURNS Pointer to new log, or **NULL** if no buffer or log list supplied, or an error occured while allocating memory.

ERRNO
S_smObjLib_NOT_INITIALIZED
S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO
wvLib, **wvLogDelete( )**

wvLogDelete( )

NAME
wvLogDelete( ) – Delete a System Viewer log

SYNOPSIS

```
STATUS wvLogDelete
    (
    WV_LOG_LIST * pLogList,  /* log list in which log appears */
    WV_LOG *      pWvLog     /* System Viewer log to delete */
    )
```

DESCRIPTION
This routine deletes a System Viewer log, and removes it from the log list. If the chosen log is the current one, and logging is enabled, the operation fails

RETURNS
OK, or **ERROR** if **NULL** log or log list supplied, or an error occured freeing memory

ERRNO
S_smObjLib_NOT_INITIALIZED
S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO
wvLib, **wvLogCreate( )**

wvLogFirstGet( )

NAME
wvLogFirstGet( ) – return a pointer to the first log in the System Viewer log list

SYNOPSIS

```
WV_LOG * wvLogFirstGet (void)
```

DESCRIPTION
This routine returns a pointer to the first System Viewer log in the list of logs. If there is no log list, the function returns **NULL**

RETURNS
Pointer to the log, or **NULL**

ERRNO

SEE ALSO
wvLib, **wvLogNextGet( )**, **logCountGet( )**

wvLogListCreate( )

NAME

wvLogListCreate( ) – create a list to hold System Viewer logs

SYNOPSIS

```
WV_LOG_LIST * wvLogListCreate (void)
```

DESCRIPTION

This routine creates a list to hold System Viewer logs. The list is created in the memory partition returned by **wvPartitionGet( )**. If the partition has not been set, or the required memory could not be allocated, the routine returns **NULL**. If a log list has already been created, the function returns **NULL**, so the old list should be deleted before a new one is created. Otherwise, it returns a pointer to the newly-created list.

RETURNS

pointer to the created **WV_LOG_LIST**, or **NULL**

ERRNO

S_smObjLib_NOT_INITIALIZED
S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO

wvLib, **wvLogListDelete( )**

wvLogListDelete( )

NAME

wvLogListDelete( ) – delete a System Viewer log list

SYNOPSIS

```
STATUS wvLogListDelete
    (
    WV_LOG_LIST * pLogList
    )
```

DESCRIPTION

This function deletes a System Viewer log list, and all its contents. If logging is enabled, it returns **ERROR**.

RETURNS

Ok, or **ERROR** if no log list supplied, or an error occured while freeing memory

ERRNO

S_smObjLib_NOT_INITIALIZED
S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO

wvLib, **wvLogListCreate( )**

wvLogNextGet( )

NAME

wvLogNextGet( ) – return a pointer to the next log in the System Viewer log list

SYNOPSIS

```
WV_LOG * wvLogNextGet
    (
    WV_LOG * pWvLog
    )
```

DESCRIPTION

This routine returns a pointer to the next System Viewer log in the list of logs.

RETURNS

Pointer to the log, or **NULL**

ERRNO

SEE ALSO

wvLib, **wvLogFirstGet( )**, **wvLogCountGet( )**

wvObjInst( )

NAME

wvObjInst( ) – instrument objects

SYNOPSIS

```
int wvObjInst
    (
    enum windObjClassType   objType,  /* object type */
    void *                  objId,    /* obj ID or NULL for all objs */
    WV_INSTRUMENTATION_MODE mode      /* instrumentation mode */
    )
```

DESCRIPTION

This routine instruments a specified object or set of objects and has effect when system objects have been enabled for event logging. *objType* could be set to any type of recognized WIND objects as listed in the **types/vxWind.h** enum list, provided they have been instrumented:

```
      windInvalidClass       = 0     invalid class type class
   1  windSemClass         *  Wind native semaphore
   2  windSemPxClass          POSIX semaphore
   3  windMsgQClass        *  Wind native message queue
   4  windMqPxClass           POSIX message queue
   5  windRtpClass         *  realtime process
   6  windTaskClass        *  task
   7  windWdClass          *  watchdog
   8  windFdClass          *  file descriptor
   9  windPgPoolClass                 page pool
   10 windPgMgrClass          page manager
   11 windGrpClass            group
   12 windVmContextClass              virtual memory context
```

```
13 windTrgClass           trigger
14 windMemPartClass    *       memory partition
15 windI2oClass           I2O
16 windDmsClass           device management system
17 windOmsClass           object management system (HA/FT)
18 windSetClass           set
19 windIsrClass        *       ISR objects
20 windTimerClass              timer services
21 windSdClass                 Shared data region
```

In the list above, the instrumented objects are marked with an asterisk. *objId* specifies the identifier of the particular object to be instrumented. If *objId* is **NULL**, then all objects of *objType* have instrumentation turned on or off depending on the value of *mode*.

If *mode* is **INSTRUMENT_ON**, instrumentation is turned on; if it is **INSTRUMENT_OFF** then instrumentation is turned off. Any other value has no effect, but the current mode (**INSTRUMENTATION_ON** or **INSTRUMENTATION_OFF**) is returned.

Use **wvSigInst( )** if you want to enable instrumentation for all signal activity, **wvEventInst( )** for vxWorks Event activity, wvSalInst for all SAL call activities.

This routine has effect only if the component **INCLUDE_WINDVIEW** is included in the project.

RETURNS **INSTRUMENT_ON**, **INSTRUMENT_OFF**, or **ERROR**.

ERRNO **S_objLib_OBJ_ID_ERROR**
S_objLib_OBJ_ILLEGAL_CLASS_TYPE

SEE ALSO **wvLib**, **wvSigInst( )**, **wvEventInst( )**, **wvObjInstAllClear( )**, **wvAllObjsSet( )**

wvObjInstModeSet( )

NAME **wvObjInstModeSet( )** – set object instrumentation on/off

SYNOPSIS

```
STATUS wvObjInstModeSet
    (
    int mode  /* object instrumentation on/off */
    )
```

DESCRIPTION This routine causes objects to be created either instrumented or not depending on the value of *mode*, which can be **INSTRUMENT_ON** or **INSTRUMENT_OFF**. All objects created after **wvObjInstModeSet( )** is called with **INSTRUMENT_ON** and before it is called with **INSTRUMENT_OFF** are created as instrumented objects.

Use **wvObjInst( )** if you want to enable instrumentation for a specific object or set of objects. Use **wvSigInst( )** if you want to enable instrumentation for all signal activity, and **wvEventInst( )** to enable instrumentation for VxWorks Event activity.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h**.

RETURNS The previous value of *mode* or **ERROR**.

ERRNO

SEE ALSO **wvLib**, **wvObjInst( )**, **wvSigInst( )**, **wvEventInst( )**, **wvEdrInst( )**

wvPartitionGet( )

NAME **wvPartitionGet( )** – determine partition in use for System Viewer logging

SYNOPSIS

```
PART_ID wvPartitionGet (void)
```

DESCRIPTION This routine returns the mamory partition id being used for System Viewer logs.

RETURNS partition id

ERRNO

SEE ALSO **wvLib**, **wvPartitionSet( )**

wvPartitionSet( )

NAME **wvPartitionSet( )** – specify a partition for use by System Viewer logging

SYNOPSIS

```
void wvPartitionSet
    (
    PART_ID memPart
    )
```

DESCRIPTION This routine allows the user to specify a memory partition to be used for System Viewer logging. Subsequent calls to create System Viewer log lists will result in the logs being created in this partition.

If using a post-mortem log, this routine should be called before a new log is created. Then the logs can be read, and a new partition and log list created, if required. Note that in post-mortem mode, if the target has rebooted, then a log list in the preserved memory should not be deleted, because the memory partition was not created during this run of the target.

RETURNS n/a

ERRNO

SEE ALSO **wvLib**, **wvPartitionGet( )**

wvRBuffMgrPrioritySet( )

NAME **wvRBuffMgrPrioritySet( )** – set the priority of the System Viewer rBuff manager

SYNOPSIS

```
STATUS wvRBuffMgrPrioritySet
    (
    int priority  /* new priority */
    )
```

DESCRIPTION This routine changes the priority of the **tWvRBuffMgr** task to the value of *priority*. Priorities range from 0, the highest priority, to 255, the lowest priority. If the task is not yet running, this priority is used when it is spawned.

RETURNS **OK**, or **ERROR** if the priority can not be set.

ERRNO Not Available

SEE ALSO **rBuffLib**, **taskPrioritySet( )**, the VxWorks programmer guides.

wvSalInst( )

NAME **wvSalInst( )** – instrument SAL

SYNOPSIS

```
int wvSalInst
    (
    WV_INSTRUMENTATION_MODE mode  /* INSTRUMENT_ON, INSTRUMENT_OFF */
    )
```

DESCRIPTION This routine instruments all SAL activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for SAL call is turned on; if it is **INSTRUMENT_OFF**, instrumentation for SAL call is turned off.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

Parameters:

mode
The required instrumentation mode. The value **INSTRUMENT_ON** enables instrumentation for SAL, **INSTRUMENT_OFF** disables it, and any other value causes the current state to be returned.

RETURNS
The mode (**INSTRUMENT_ON** or **INSTRUMENT_OFF**) currently in force.

ERRNO
Not Available

SEE ALSO
wvLib

wvSigInst( )

NAME
wvSigInst( ) – instrument signals

SYNOPSIS
```
int wvSigInst
    (
    WV_INSTRUMENTATION_MODE mode  /* INSTRUMENT_ON, INSTRUMENT_OFF */
    )
```

DESCRIPTION
This routine instruments all signal activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for signals is turned on; if it is **INSTRUMENT_OFF**, instrumentation for signals is turned off.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

Parameters:

mode
The required instrumentation mode. The value **INSTRUMENT_ON** enables instrumentation for signals, **INSTRUMENT_OFF** disables it, and any other value causes the current state to be returned.

RETURNS
The mode (**INSTRUMENT_ON** or **INSTRUMENT_OFF**) currently in force.

ERRNO

SEE ALSO **wvLib**

wvSockUploadPathClose( )

NAME **wvSockUploadPathClose( )** – close the socket upload path

SYNOPSIS

```
void wvSockUploadPathClose
    (
    UPLOAD_ID upId  /* generic upload-path descriptor */
    )
```

DESCRIPTION This routine closes the socket connection to the event receiver on the host.

RETURNS N/A

ERRNO Not Available

SEE ALSO **wvSockUploadPathLib**, **sockUploadPathCreate( )**

wvSockUploadPathCreate( )

NAME **wvSockUploadPathCreate( )** – establish an upload path to the host using a socket

SYNOPSIS

```
UPLOAD_ID wvSockUploadPathCreate
    (
    char  *ipAddress,  /* server's hostname or IP address in .-notation */
    short port         /* port number to bind to */
    )
```

DESCRIPTION This routine initializes the TCP/IP connection to the host process that receives uploaded events. It can be retried if the connection attempt fails.

RETURNS The **UPLOAD_ID**, or **NULL** if the connection cannot be completed or memory for the ID is not available.

ERRNO Not Available

SEE ALSO **wvSockUploadPathLib**, **sockUploadPathClose( )**

wvSockUploadPathLibInit( )

NAME **wvSockUploadPathLibInit( )** – initialize **wvSockUploadPathLib** library

SYNOPSIS

```
STATUS wvSockUploadPathLibInit (void)
```

DESCRIPTION This routine initializes **wvSockUploadPathLib** by pulling in the routines in this file for use with Wind River System Viewer. It is called during system configuration from **usrWindview.c**.

RETURN **OK**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **wvSockUploadPathLib**

wvSockUploadPathWrite( )

NAME **wvSockUploadPathWrite( )** – write to the socket upload path

SYNOPSIS

```
int wvSockUploadPathWrite
    (
    UPLOAD_ID upId,     /* generic upload-path descriptor */
    char *    pStart,   /* address of data to write */
    size_t    size      /* number of bytes of data at pStart */
    )
```

DESCRIPTION This routine writes *size* bytes of data beginning at *pStart* to the upload path between the target and the event receiver on the host.

RETURNS The number of bytes written, or **ERROR**.

ERRNO Not Available

SEE ALSO **wvSockUploadPathLib**, **wvSockUploadPathCreate( )**

wvTmrRegister( )

NAME

wvTmrRegister( ) – register a timestamp timer

SYNOPSIS

```
void wvTmrRegister
    (
    UINTFUNCPTR wvTmrRtn,       /* timestamp routine */
    UINTFUNCPTR wvTmrLockRtn,   /* locked timestamp routine */
    FUNCPTR     wvTmrEnable,    /* enable timer routine */
    FUNCPTR     wvTmrDisable,   /* disable timer routine */
    FUNCPTR     wvTmrConnect,   /* connect to timer routine */
    UINTFUNCPTR wvTmrPeriod,    /* period of timer routine */
    UINTFUNCPTR wvTmrFreq       /* frequency of timer routine */
    )
```

DESCRIPTION

This routine registers a timestamp routine for each of the following:

wvTmrRtn
a timestamp routine, which returns a timestamp when called (must be called with interrupts locked).

wvTmrLockRtn
a timestamp routine, which returns a timestamp when called (locks interrupts).

wvTmrEnable
an enable-timer routine, which enables the timestamp timer.

wvTmrDisable
a disable-timer routine, which disables the timestamp timer.

wvTmrConnect
a connect-to-timer routine, which connects a handler to be run when the timer rolls over; this routine should return **ERROR** if the system clock tick is to be used.

wvTmrPeriod
a period-of-timer routine, which returns the period of the timer.

wvTmrFreq
a frequency-of-timer routine, which returns the frequency of the timer.

If any of these routines is set to **NULL**, the behavior of instrumented code is undefined.

RETURNS

N/A

ERRNO

SEE ALSO

wvTmrLib

wvTsfsUploadPathClose( )

NAME

wvTsfsUploadPathClose( ) – close the TSFS-socket upload path

SYNOPSIS

```
void wvTsfsUploadPathClose
    (
    UPLOAD_ID upId  /* generic upload-path descriptor */
    )
```

DESCRIPTION

This routine closes the TSFS-socket connection to the event receiver on the host.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

wvTsfsUploadPathLib, **wvTsfsUploadPathCreate( )**

wvTsfsUploadPathCreate( )

NAME

wvTsfsUploadPathCreate( ) – open an upload path to the host using a TSFS socket

SYNOPSIS

```
UPLOAD_ID wvTsfsUploadPathCreate
    (
    char  *ipAddress,  /* server's IP address in .-notation */
    short port         /* port number to bind to */
    )
```

DESCRIPTION

This routine opens a TSFS socket to the host to be used for uploading event data. After successfully establishing this connection, an **UPLOAD_ID** is returned which points to the **TSFS_UPLOAD_DESC** that is passed to **open( )**, **close( )**, **read( )**, etc. for future operations.

RETURNS

The **UPLOAD_ID**, or **NULL** if the connection cannot be completed or not enough memory is available.

ERRNO

Not Available

SEE ALSO

wvTsfsUploadPathLib, **wvTsfsUploadPathClose( )**

wvTsfsUploadPathLibInit( )

NAME **wvTsfsUploadPathLibInit( )** – initialize **wvTsfsUploadPathLib** library

SYNOPSIS
```
STATUS wvTsfsUploadPathLibInit (void)
```

DESCRIPTION This routine initializes **wvTsfsUploadPathLib** by pulling in the routines in this file for use with the Wind River System Viewer. It is called during system configuration from **usrWindview.c**.

RETURNS **OK**.

ERRNO Not Available

SEE ALSO **wvTsfsUploadPathLib**

wvTsfsUploadPathWrite( )

NAME **wvTsfsUploadPathWrite( )** – write to the TSFS upload path

SYNOPSIS
```
int wvTsfsUploadPathWrite
    (
    UPLOAD_ID upId,    /* generic upload-path descriptor */
    char *    pStart,  /* address of data to write */
    size_t    size     /* number of bytes of data at pStart */
    )
```

DESCRIPTION This routine writes *size* bytes of data beginning at *pStart* to the upload path connecting the target with the host receiver.

RETURNS The number of bytes written, or **ERROR**.

ERRNO Not Available

SEE ALSO **wvTsfsUploadPathLib**, **wvTsfsUploadPathCreate( )**

wvUploadStart( )

NAME

wvUploadStart( ) – start upload of events to the host

SYNOPSIS

```
WV_UPLOADTASK_ID wvUploadStart
    (
    WV_LOG *  pWvLog,             /* System Viewer log */
    UPLOAD_ID pathId,             /* upload path to host */
    BOOL      uploadContinuously  /* upload continuously if true */
    )
```

DESCRIPTION

This routine starts uploading events from the System Viewer log to the host. Events can be uploaded either continuously or in one pass until the log is emptied. If *uploadContinuously* is set to **TRUE**, the task uploading events pends until more data arrives in the buffer. If **FALSE**, the buffer is flushed without waiting, but this routine returns immediately with an ID that can be used to kill the upload task. Upload is done by spawning the task **tWVUpload**. The log to upload is identified by *pWvLog*, and the upload path to use is identified by *pathId*.

This routine blocks if no event data is in the buffer, so it should be called before event logging is started to ensure the buffer does not overflow.

RETURNS

A valid **WV_UPLOADTASK_ID** if started for continuous upload, a non-**NULL** value if started for one-pass upload, and **NULL** if the task can not be spawned or memory for the descriptor can not be allocated.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO

wvLib

wvUploadStop( )

NAME

wvUploadStop( ) – stop upload of events to host

SYNOPSIS

```
STATUS wvUploadStop
    (
    WV_UPLOADTASK_ID upTaskId
    )
```

DESCRIPTION

This routine stops continuous upload of events to the host. It does this by making a request to the upload task to terminate after it has emptied the buffer. For this reason it is important to make sure data is no longer being logged to the buffer before calling this routine.

This task blocks until the buffer is emptied, and then frees memory associated with *upTaskId*.

RETURNS

OK if the upload task terminates successfully, or **ERROR** either if *upTaskId* is invalid or if the upload task terminates with an **ERROR**.

ERRNO

Not Available

SEE ALSO

wvLib

wvUploadTaskConfig( )

NAME

wvUploadTaskConfig( ) – set priority and stacksize of **tWVUpload** task

SYNOPSIS

```
void wvUploadTaskConfig
    (
    int stackSize,  /* the new stack size for tWVUpload */
    int priority    /* the new priority for tWVUpload */
    )
```

DESCRIPTION

This routine sets the stack size and priority of future instances of the event-data upload task, created by calling **wvUploadStart( )**. The default stack size for this task is 5000 bytes, and the default priority is 150.

RETURNS

N/A

ERRNO

SEE ALSO

wvLib

xattrib( )

NAME

xattrib( ) – modify MS-DOS file attributes of many files

SYNOPSIS

```
STATUS xattrib
    (
    const char * source,  /* file or dir name on which to change flags */
    const char * attr     /* flag settings to change */
    )
```

DESCRIPTION This function is essentially the same as **attrib( )**, but it accepts wildcards in *fileName*, and traverses subdirectories in order to modify attributes of entire file hierarchies.

The *attr* argument string may contain must start with either "+" or "-", meaning the attribute flags which will follow should be either set or cleared. After "+" or "-" any of these four letter will signify their respective attribute flags - "A", "S", "H" and "R".

EXAMPLE

```
-> xattrib( "/sd0/sysfiles", "+RS") /* write protect "sysfiles" */
-> xattrib( "/sd0/logfiles", "-R")  /* unprotect logfiles before deletion */
-> xdelete( "/sd0/logfiles")
```

CAVEAT This function may call itself in accordance with the depth of the source directory, and allocates 2 kB of heap memory per stack frame, meaning that to accommodate the maximum depth of subdirectories which is 20, at least 40 kB of heap memory should be available.

RETURNS **OK**, or **ERROR** if the file can not be opened.

ERRNO Not Available

SEE ALSO **usrFsLib**, **dosFsLib**, the VxWorks programmer guides.

xbdBlkDevCreate( )

NAME **xbdBlkDevCreate( )** – create an XBD block device wrapper

SYNOPSIS

```
device_t xbdBlkDevCreate
    (
    BLK_DEV *    bd,   /* pointer to block device */
    const char * name  /* pointer to device name  */
    )
```

DESCRIPTION This routine creates an XBD block device wrapper.

RETURNS a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO **xbdBlkDev**

xbdBlkDevCreateSync( )

NAME **xbdBlkDevCreateSync( )** – synchronously create an XBD block device wrapper

SYNOPSIS

```
device_t xbdBlkDevCreateSync
    (
    BLK_DEV *    bd,   /* pointer to block device */
    const char * name  /* pointer to device name  */
    )
```

DESCRIPTION This routine creates an XBD block device wrapper. It returns after the entire XBD stack has been created/initialized.

RETURNS a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO **xbdBlkDev**

xbdBlkDevDelete( )

NAME **xbdBlkDevDelete( )** – deletes an XBD block device wrapper

SYNOPSIS

```
STATUS xbdBlkDevDelete
    (
    device_t   d,    /* device_t returned from xbdBlkDevCreate */
    BLK_DEV ** ppbd  /* pointer to block device pointer */
    )
```

DESCRIPTION This routine deletes or destroys an XBD block device wrapper.

The *d* parameter specifies the XBD block wrapper to delete. This should be the same value that was returned from **xbdBlkDevCreate( )**

The *ppbd* parameter is an out parameter that can be used to return the block device pointer used in **xbdBlkDevCreate( )**. If specified as **NULL** no attempt to return the block device pointer is attempted.

RETURNS a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO **xbdBlkDev**

xbdBlkDevLibInit( )

NAME **xbdBlkDevLibInit( )** – initialize the XBD block device wrapper

SYNOPSIS

```
STATUS xbdBlkDevLibInit
    (
    int xbdServiceTskPri
    )
```

DESCRIPTION This routine initializes the XBD block device wrapper.

RETURNS **OK**

ERRNO N/A

SEE ALSO **xbdBlkDev**

xbdCbioDevCreate( )

NAME **xbdCbioDevCreate( )** – create an XBD CBIO device wrapper

SYNOPSIS

```
device_t xbdCbioDevCreate
    (
    CBIO_DEV_ID  cbio,  /* handle to CBIO device */
    const char * name,  /* pointer to device name  */
    unsigned int opts   /* options for device */
    )
```

DESCRIPTION This routine creates an XBD CBIO device wrapper. It returns after the entire XBD stack has been created/initialized.

cbio handle to previously created CBIO device

name base name of the XBD/CBIO wrapper

The *opts* argument is a bit-wise or'ed combination of options controlling the operation of this routine as follows:

XBD_CBIO_NOWAIT
Function will not wait until path(s) are instantiated and will return immediately

XBD_CBIO_NOPART
Wrapper is not capabable of supporting partitions. Device will be viewed as one partition spanning the entire media.

XBD_CBIO_DEFAULT
The default behaviour. Partitions are supported and xbdCbioDevCreate will not return until all paths are instantiated.

RETURNS a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO **xbdCbioDev**

xbdCbioDevDelete( )

NAME **xbdCbioDevDelete( )** – deletes an XBD CBIO device wrapper

SYNOPSIS

```
STATUS xbdCbioDelete
    (
    device_t     d,     /* device_t returned from xbdCbioDevCreate */
    CBIO_DEV_ID* pCbio  /* pointer to CBIO handle */
    )
```

DESCRIPTION This routine deletes or destroys an XBD CBIO device wrapper.

The *d* parameter specifies the XBD CBIO wrapper to delete. This should be the same value that was returned from **xbdCbioDevCreate( )**

The *ppbd* parameter is an out parameter that can be used to return the CBIO handle used in **xbdCbioDevCreate( )**. If specified as **NULL** no attempt to return the handle is attempted.

RETURNS **OK** upon success, or **ERROR** otherwise

ERRNO

SEE ALSO **xbdCbioDev**

xbdCbioLibInit( )

NAME **xbdCbioLibInit( )** – initialize the XBD block device wrapper

SYNOPSIS

```
STATUS xbdCbioDevLibInit (void)
```

DESCRIPTION This routine initializes the XBD block device wrapper.

RETURNS **OK**

ERRNO N/A

SEE ALSO **xbdCbioDev**

xbdCreatePartition( )

NAME **xbdCreatePartition( )** – partition an XBD device

SYNOPSIS

```
STATUS xbdCreatePartition
    (
    char *pathName,  /* name of device to partition */
    int  nPart,      /* number of partitions */
    int  size1,      /* space percentage for second partition */
    int  size2,      /* space percentage for third partition */
    int  size3       /* space percentage for fourth partition */
    )
```

DESCRIPTION This function is capable of creating only one partition table - the MBR, and will not create any Bootable or Extended partitions. Therefore, only 4 primary partitions are supported.

pathName is the name the device to be partitioned.

The *nPart* argument contains the number of partitions to create. If *nPart* is 0 or 1, a single partition covering the entire disk is created. If *nPart* is between 2 and 4, the arguments *size1*, *size2* and *size3* contain (as integers) the *percentage* of disk space to be assigned to the 2nd, 3rd, and 4th partitions respectively. The first partition (partition 0) will be assigned the remaining space. Thus, the sum of the three sizes should be less than 100.

Partition sizes will be rounded down to be multiple of whole tracks so that partition Cylinder/Head/Track fields will be initialized as well as the LBA fields. Although the CHS fields are written they are not used in VxWorks, and can not be guaranteed to work correctly on other systems.

RETURNS **OK** upon success, **ERROR** otherwise

ERRNO Not Available

SEE ALSO **partLib**

xbdRamDiskDevCreate( )

NAME **xbdRamDiskDevCreate( )** – create an XBD ram disk

SYNOPSIS

```
device_t xbdRamDiskDevCreate
    (
    unsigned     blockSize,  /* block size in bytes */
    unsigned     totalSize,  /* disk size in bytes */
    BOOL         flag,       /* should the disk support partitions? */
    const char * name        /* name of ram disk */
    )
```

DESCRIPTION This routine creates an XBD ram disk. The ram disk links into the file system monitor and eventing framework.

RETURNS The ID of the XBD created(device_t) or NULLDEV if the routine fails

ERRNO Not Available

SEE ALSO **xbdRamDisk**

xbdRamDiskDevDelete( )

NAME **xbdRamDiskDevDelete( )** – XBD Ram Disk Deletion routine

SYNOPSIS

```
STATUS xbdRamDiskDevDelete
    (
    device_t d  /* device_t returned from xbdRamDiskDevCreate */
    )
```

DESCRIPTION This routine deletes or destroy an instantion of an XBD ram disk. The ram disk to be deleted is identified by the supplied device_t. This value must have been previously returned from the corresponding xbdRamDiskDevCreate function. All resource associated with the ram disk are freed. Any file systems sitting on top of the ram disk are ejected.

RETURNS **OK** on success or **ERROR** if the supplied device_t doesn't map to an existing and valid XBD.

ERRNO Not Available

SEE ALSO **xbdRamDisk**

xbdTransDevCreate( )

NAME **xbdTransDevCreate( )** – create a transactional XBD.

SYNOPSIS

```
device_t xbdTransDevCreate
    (
    device_t subDev  /* lower level device */
    )
```

RETURNS device_t, or NULLDEV on failure.

ERRNO Not Available

SEE ALSO **xbdTrans**, **dosFsDevCreate( )**.

xbdTransInit( )

NAME **xbdTransInit( )** – initialize the transactional XBD subsystem.

SYNOPSIS

```
STATUS xbdTransInit
    (
    void
    )
```

DESCRIPTION We just plug ourselves in to the file system monitor so that we get called to probe partitions as they are instantiated.

RETURNS **OK** if all went well, **ERROR** otherwise.

ERRNO N/A

SEE ALSO **xbdTrans**

xcopy( )

NAME **xcopy( )** – copy a hierarchy of files with wildcards

SYNOPSIS

```
STATUS xcopy
```

```
    (
    const char * source,  /* source directory or wildcard name */
    const char * dest     /* destination directory */
    )
```

DESCRIPTION *source* is a string containing a name of a directory, or a wildcard or both which will cause this function to make a recursive copy of all files residing in that directory and matching the wildcard pattern into the *dest* directory, preserving the file names and subdirectories.

CAVEAT This function may call itself in accordance with the depth of the source directory, and allocates 3 kB of heap memory per stack frame, meaning that to accommodate the maximum depth of subdirectories which is 20, at least 60 kB of heap memory should be available.

RETURNS **OK**, or **ERROR** if any operation has failed.

ERRNO Not Available

SEE ALSO **usrFsLib**, **tarLib**, **cp( )**, the VxWorks programmer guides.

xdelete( )

NAME **xdelete( )** – delete a hierarchy of files with wildcards

SYNOPSIS

```
STATUS xdelete
    (
    const char * source  /* source directory or wildcard name */
    )
```

DESCRIPTION *source* is a string containing a name of a directory, or a wildcard or both which will cause this function to recursively remove all files and subdirectories residing in that directory and matching the wildcard pattern. When a directory is encountered, all its contents are removed, and then the directory itself is deleted.

Note that the wildcard matching is limited to a single directory level.

```
    dir         is valid
    *.c         is valid
    dir/*.c     is valid
    *a/*.c      is not valid
```

RETURNS **OK** or **ERROR** if any operation has failed.

ERRNO Not Available

SEE ALSO **usrFsLib**, **cp( )**, **copy( )**, **xcopy( )**, **tarLib**, the VxWorks programmer guides.

y( )

NAME

y( ) – return the contents of the y register (SimSolaris)

SYNOPSIS

```
int y
    (
    int taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of the y register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS

The contents of the y register.

ERRNO

Not Available

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

ykRegister( )

NAME

ykRegister( ) – register with the VxBus subsystem

SYNOPSIS

```
void ykRegister(void)
```

DESCRIPTION

This routine registers the Template driver with VxBus as a child of the PCI bus type.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

mvYukonVxbEnd

ynRegister( )

NAME

ynRegister( ) – register with the VxBus subsystem

SYNOPSIS

```
void ynRegister(void)
```

DESCRIPTION	This routine registers the Yukon II driver with VxBus as a child of the PCI bus type.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	**mvYukonIIVxbEnd**