WIND RIVER

VxWorks®

APPLICATION API REFERENCE

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 Application API Reference, 6.6

16 Nov 07
Part #: DOC-16103-ND-00

Contents

This book provides reference entries that describe the facilities available for VxWorks process-based application development. For reference entries that describe facilities available in the VxWorks kernel, see the *VxWorks Kernel API Reference*. For reference entries that describe VxWorks drivers, see the *VxWorks Drivers API Reference*.

1. Libraries

This section provides reference entries for each of the VxWorks application 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 section 2.

2. Routines

This section provides reference entries for each of the routines found in the VxWorks application libraries documented in section 1.

1
Libraries

1

aioPxLib

NAME

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

ROUTINES

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

DESCRIPTION

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

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

AIO ENVIRONMENT VARIABLES

The following environment variables can be set when loading an RTP to configure the AIO library.

ENVIRONMENT VARIABLE	DETAILS	DEFAULT SETTING (defined in .h files)
MAX_LIO_CALLS	Maximum outstanding LIO calls	**AIO_CLUST_MAX**
MAX_AIO_SYS_TASKS	Number of tasks spawned to support AIO	**AIO_IO_TASKS_DFLT**
AIO_TASK_PRIORITY	AIO tasks' priority	**AIO_IO_PRIO_DFLT**
AIO_TASK_STACK_SIZE	AIO tasks' stack size	**AIO_IO_STACK_DFLT**

AIO COMMANDS

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

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

aio_read( )
initiates an asynchronous read

aio_write( )
initiates an asynchronous write

lio_listio( )
initiates a list of asynchronous I/O requests

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

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

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

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

AIO CONTROL BLOCK

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

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

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

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

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

aio_fildes
file descriptor for I/O.

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

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

aio_nbytes
number of bytes to read or write.

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

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

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

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

EXAMPLES

A writer could be implemented as follows:

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

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

aio_write (pAioWrite);

/*  .
 .
```

```
     do other work
     .
     .
    */

    /* now wait until I/O finishes */

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

    aio_return (pAioWrite);
    free (pAioWrite);
```

A reader could be implemented as follows:

```
    /* initialize signal handler */

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

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

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

    aio_read (pAioRead);

    /*
        .
        .

        do other work
     .
     .
    */
```

The signal handler might look like the following:

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

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

INCLUDE FILES **aio.h**

SEE ALSO POSIX 1003.1b document

bLib

NAME **bLib** – buffer manipulation library

ROUTINES
bcmp( ) – compare one buffer to another
binvert( ) – invert the order of bytes in a buffer
bswap( ) – swap buffers
swab( ) – swap bytes
uswab( ) – swap bytes with buffers that are not necessarily aligned
bzero( ) – zero out a buffer
bcopy( ) – copy one buffer to another
bcopyBytes( ) – copy one buffer to another one byte at a time
bcopyWords( ) – copy one buffer to another one word at a time
bcopyLongs( ) – copy one buffer to another one long word at a time
bfill( ) – fill a buffer with a specified character
bfillBytes( ) – fill buffer with a specified character one byte at a time
index( ) – find the first occurrence of a character in a string
rindex( ) – find the last occurrence of a character in a string

DESCRIPTION This library contains routines to manipulate buffers of variable-length byte arrays. Operations are performed on long words when possible, even though the buffer lengths are specified in bytes. This occurs only when source and destination buffers start on addresses that are both odd or both even. If one buffer is even and the other is odd, operations must be done one byte at a time, thereby slowing down the process.

Certain applications, such as byte-wide memory-mapped peripherals, may require that only byte operations be performed. For this purpose, the routines **bcopyBytes( )** and **bfillBytes( )** provide the same functions as **bcopy( )** and **bfill( )**, but use only byte-at-a-time operations. These routines do not check for null termination.

INCLUDE FILES **strings.h**

SEE ALSO **ansiString**

cacheLib

NAME **cacheLib** – cache management library for processes

ROUTINES
cacheFlush( ) – flush all or some of a specified cache
cacheInvalidate( ) – invalidate all or some of a specified cache
cacheClear( ) – clear all or some entries from a cache
cacheTextUpdate( ) – synchronize the instruction and data caches

DESCRIPTION This library provides architecture-independent routines for managing the instruction and data caches in processes (RTPs). The routines provided in this library can be used to implement applications and libraries that must ensure cache coherency - such as user-level device drivers and loaders.

The routines provided in this library are expected to work only with memory that is part of the process (RTP) context, such as process heap, mapped memory, and shared data regions opened by the RTP. A process is not allowed to perform operations on the entire cache.

INCLUDE FILES **cacheLib.h**

SEE ALSO **mmanLib**

clockLib

NAME **clockLib** – user-side clock library (POSIX)

ROUTINES
clock_getres( ) – get the clock resolution (POSIX)
clock_setres( ) – set the clock resolution
clock_gettime( ) – get the current time of the clock (POSIX)
clock_settime( ) – set the clock to a specified time (POSIX)
clock_nanosleep( ) – high resolution sleep with specifiable clock

DESCRIPTION This library provides a clock interface, as defined in the IEEE standard, POSIX 1003.1b.

A clock is a software construct that keeps time in seconds and nanoseconds. The clock has a simple interface with three routines: **clock_settime( )**, **clock_gettime( )**, and **clock_getres( )**. The non-POSIX routine **clock_setres( )** that was provided so that **clockLib** could be informed if there were changes in the system clock rate is no longer necessary. This routine is still present for backward compatibility, but does nothing.

Times used in these routines are stored in the timespec structure:

```
struct timespec
```

```
{
time_t        tv_sec;          /* seconds */
long  tv_nsec;        /* nanoseconds (0 -1,000,000,000) */
};
```

IMPLEMENTATION

The required *clock_id* values **CLOCK_REALTIME**, **CLOCK_MONOTONIC** and **CLOCK_THREAD_CPUTIME_ID** are supported - the value returned by the **pthread_getcpuclockid( )** API can be used as the *clock_id*.

The *clock_id* **CLOCK_PROCESS_CPUTIME_ID** is NOT supported.

Conceivably, additional "virtual" clocks could be supported, or support for additional auxiliary clock hardware (if available) could be added.

CONFIGURATION

This library requires the **INCLUDE_POSIX_CLOCKS** component to be configured into the kernel; *errno* may be set to **ENOSYS** if this component is not present.

INCLUDE FILES

time.h

SEE ALSO

IEEE POSIX 1003.1b documentation

confstr

NAME

confstr – POSIX 1003.1/1003.13 (PSE52) **confstr( )** API

ROUTINES

confstr( ) – get strings associated with system variables

DESCRIPTION

This module contains the POSIX conforming **confstr( )** routine used by applications to get the values of configuration-defined variables which store strings.

INCLUDE FILES

unistd.h

cpuset

NAME

cpuset – cpuset_t type manipulation macros

ROUTINES

CPUSET_SET( ) – set a CPU in a CPU set
CPUSET_CLR( ) – clear a CPU from a CPU set
CPUSET_ZERO( ) – clear all CPUs from a CPU set
CPUSET_ISSET( ) – determine if a CPU is set in a CPU set

CPUSET_ISZERO( ) – determine if all CPUs are cleared from a CPU set
CPUSET_ATOMICSET( ) – atomically set a CPU in a CPU set
CPUSET_ATOMICCLR( ) – atomically clear a CPU from a CPU set
CPUSET_ATOMICCOPY( ) – atomically copy a CPU set value

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

INCLUDE FILES **cpuset.h**

SEE ALSO **vxCpuLib**

dirLib

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

ROUTINES
opendir( ) – open a directory for searching (POSIX)
readdir( ) – read one entry from a directory (POSIX)
readdir_r( ) – read one entry from a directory (POSIX)
rewinddir( ) – reset position to the start of a directory (POSIX)
closedir( ) – close a directory (POSIX)
fstat( ) – get file status information (POSIX)
stat( ) – get file status information using a pathname (POSIX)
fstatfs( ) – get file status information (POSIX)
statfs( ) – get file status information using a pathname (POSIX)
utime( ) – update time on a file

DESCRIPTION This library provides POSIX-defined routines for opening, reading, and closing directories on a file system. It also provides routines to obtain more detailed information on a file or directory.

CONFIGURATION To use the POSIX directory-handling library, configure VxWorks with the **INCLUDE_POSIX_DIRLIB** component.

SEARCHING DIRECTORIES

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

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

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

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

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

GETTING FILE INFORMATION

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

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

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

S_ISREG
: test if the file is a regular file

S_ISDIR
: test if the file is a directory

S_ISCHR
: test if the file is a character special file

S_ISBLK
: test if the file is a block special file

S_ISFIFO
: test if the file is a FIFO special file

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

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

```
    char          *filename;
    struct stat   fileStat;

    stat (filename, &fileStat);

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

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

INCLUDE FILES

dirent.h, **stat.h**

edrLib

NAME

edrLib – Error Detection and Reporting subsystem

ROUTINES

edrErrorInject( ) – injects an error into the ED&R subsystem
edrIsDebugMode( ) – determines if the ED&R debug flag is set
_edrErrorInject( ) – inject an ED&R error record (system call)
edrFlagsGet( ) – return the current ED&R flags set in the kernel (system call)

DESCRIPTION

This library provides the user level public API for the ED&R subsystem, covering error injection and status information. See the kernel **edrLib** documentation for a complete description of the ED&R facilities.

INCLUDE FILES

edrLib.h

errnoLib

NAME

errnoLib – user-level error status library

ROUTINES

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

DESCRIPTION

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

This facility is compatible with the UNIX error status mechanism in which error status values are set in what appears to be a global variable **errno**. However, in VxWorks there are many tasks in an RTP that share common memory space and therefore would conflict if **errno** were really a global variable.

VxWorks resolves this by maintaining the **errno** value for each context separately in the task's TCB.

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

An error status is a 4-byte integer. By convention, the most significant two bytes are the module number, which indicates the module in which the error occurred. The lower two bytes indicate the specific error within that module. Module number 0 is reserved for UNIX error numbers so that values from the UNIX **errno.h** header file can be set and tested without modification. Module numbers 1-500 decimal are reserved for VxWorks modules. These are defined in **vwModNum.h**. All other module numbers are available to applications.

VxWorks can include a special symbol table in the kernel called **statSymTbl** which **printErrno( )** uses to print human-readable error messages. See the kernel **errnoLib** reference entry for more details regarding **statSymTbl**.

INCLUDE FILES

The file **vwModNum.h** contains the module numbers for every VxWorks module., The include file for each module contains the error numbers which that module, can generate.

SEE ALSO

printErrno( ) (kernel), **makeStatTbl** (kernel)

eventLib

NAME

eventLib – VxWorks user events library

ROUTINES

eventClear( ) – Clear all events for calling task
eventReceive( ) – Receive event(s) for the calling task
eventSend( ) – Send event(s) to a task

DESCRIPTION

Events are a means of communication between tasks based on a synchronous model. Only tasks can receive events while tasks, semaphore and message queue can send.

Events are similar to signals in that they are sent to a task asynchronously. But differ in that it is synchronous in receiving. i.e., the receiving task must call a function to receive at will and can choose to pend when waiting for events to arrive. Thus, unlike signals, event handler is not implemented.

Each task has its own events field that can be filled by having tasks (even itself) and semaphore and message queue sending events to the task when they are available. Each event's meaning is different for every task. Event X when received can be interpreted differently by separate tasks. Also, it should be noted that events are not accumulated. If the same event is received several times, it counts as if it were received only once. It is not possible to track how many times each event has been sent to a task.

There are some VxWorks objects that can send events when they become available. They are referred to as **resources** in the context of events. They include semaphores and message queues. For example, when a semaphore becomes free, events can be sent to a task that asked for it.

This file implements user event feature which is events sent, received in a RTP, across RTPs or between RTP and kernel tasks; And events sent by user semaphore or user msgQ and received by RTPs.

INCLUDE FILES **eventLib.h**

SEE ALSO **taskLib**, **semEvLib**, **msgQEvLib**, the VxWorks programmer guides.

ffsLib

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

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

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

INCLUDE FILES **ffsLib.h**

fioLib

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

ROUTINES

oprintf( ) – write a formatted string to an output function
voprintf( ) – write a formatted string to an output function
printErr( ) – write a formatted string to the standard error stream
fdprintf( ) – write a formatted string to a file descriptor
vfdprintf( ) – write a string formatted with a variable argument list to a file descriptor
fioFormatV( ) – convert a format string
fioRead( ) – read a buffer
fioRdString( ) – read a string from a file

DESCRIPTION

This library provides the basic formatting and scanning I/O functions. These are Non-ANSI routines.

INCLUDE FILES

none

SEE ALSO

libc

---

fsPxLib

NAME

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

ROUTINES

unlink( ) – unlink a file
link( ) – link a file
fsync( ) – synchronize a file
fdatasync( ) – synchronize a file data
rename( ) – change the name of a file
fpathconf( ) – determine the current value of a configurable limit
pathconf( ) – determine the current value of a configurable limit
access( ) – determine accessibility of a file
fcntl( ) – perform control functions over open files
chmod( ) – change the permission mode of a file
fchmod( ) – change the permission mode of a file

DESCRIPTION

This library contains POSIX APIs which are applicable to I/O and the file system.

INCLUDE FILES

ioLib.h, **stdio.h**

SEE ALSO

ioLib, **iosLib**, the VxWorks programmer guides.

ftruncate

NAME
ftruncate – POSIX file truncation

ROUTINES
ftruncate( ) – truncate a file (POSIX)

DESCRIPTION
This module contains the POSIX compliant **ftruncate( )** routine for truncating a file.

INCLUDE FILES
unistd.h

SEE ALSO

getenv

NAME
getenv – POSIX environment variable **getenv( )** routine

ROUTINES
getenv( ) – get value of an environment variable (POSIX)

DESCRIPTION
This module contains the POSIX compliant **getenv( )** routine to get the value of environment variables from the RTP's environment.

Although this routine is thread-safe, if the application directly modifies the *environ* array or the pointer to which it points, the behavior of **getenv( )** is undefined.

INCLUDE FILES
stdlib.h

getopt

NAME
getopt – getopt facility

ROUTINES
getopt( ) – parse argc/argv argument vector (POSIX)
getoptInit( ) – initialize the getopt state structure
getopt_r( ) – parse argc/argv argument vector (POSIX)
getOptServ( ) – parse parameter string into argc, argv format

DESCRIPTION
This library supplies both a POSIX compliant **getopt( )** which is a command line parser, as well as a rentrant version of the same command named **getopt_r( )**. Prior to calling **getopt_r( )**, the caller needs to initialize the getopt state structure by calling **getoptInit( )**.

This explicit initialization is not needed while calling **getopt( )** as the system is setup as if the initialization has already been done.

The user can modify **getopt( )** behavior by setting the the getopt variables like optind, opterr, etc. For **getopt_r( )**, the value needs to be updated in the getopt state structure.

INCLUDE FILES none

hashLib

NAME **hashLib** – generic hashing library

ROUTINES

hashTblCreate( ) – create a hash table
hashTblInit( ) – initialize a hash table
hashTblDelete( ) – delete a hash table
hashTblTerminate( ) – terminate a hash table
hashTblDestroy( ) – destroy a hash table
hashTblPut( ) – put a hash node into the specified hash table
hashTblFind( ) – find a hash node that matches the specified key
hashTblRemove( ) – remove a hash node from a hash table
hashTblEach( ) – call a routine for each node in a hash table
hashFuncIterScale( ) – iterative scaling hashing function for strings
hashFuncModulo( ) – hashing function using remainder technique
hashFuncMultiply( ) – multiplicative hashing function
hashKeyCmp( ) – compare keys as 32 bit identifiers
hashKeyStrCmp( ) – compare keys based on strings they point to

DESCRIPTION This subroutine library supports the creation and maintenance of a chained hash table. Hash tables efficiently store hash nodes for fast access. They are frequently used for symbol tables, or other name to identifier functions. A chained hash table is an array of singly linked list heads, with one list head per element of the hash table. During creation, a hash table is passed two user-definable functions, the hashing function, and the hash node comparator.

CONFIGURATION To use the generic hashing library, configure VxWorks with the **INCLUDE_HASH** component.

HASH NODES A hash node is a structure used for chaining nodes together in the table. The defined structure **HASH_NODE** is not complete because it contains no field for the key for referencing, and no place to store data. The user completes the hash node by including a **HASH_NODE** in a structure containing the necessary key and data fields. This flexibility allows hash tables to better suit varying data representations of the key and data fields. The hashing function and the hash node comparator determine the full hash node representation. Refer to the defined structures **H_NODE_INT** and **H_NODE_STRING** for examples of the general purpose hash nodes used by the hashing functions and hash node comparators defined in this library.

HASHING FUNCTIONS

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

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

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

HASH NODE COMPARATOR FUNCTIONS

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

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

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

```
BOOL hashKeyCmpXXX
```

1

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

HASHING COLLISIONS

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

STRUCTURE

```
HASH_HEAD 0           HASH_NODE         HASH_NODE
---------             --------          --------
| head--------------->| next----------->| next---------
|       |             |......|          |......|       |
| tail------          | key  |          | key  |       |
|       |  |          | data |          | data |       v
---------  |          --------          --------      ---
           |                             ^             -
           |                             |
           -------------------------------

HASH_HEAD 1           HASH_NODE
---------             --------
| head--------------->| next---------
|       |             |......|       |
| tail------          | key  |       |
|       |  |          | data |       v
---------  |          --------      ---
           |           ^             -
           |           |
           -------------

 ...
 ...

HASH_HEAD N

---------
| head-----------------
|       |             |
| tail---------       |
|       |     |       v
---------    ---     ---
        -       -
```

CAVEATS

Hash tables must have a number of elements equal to a power of two.

INCLUDE FILE **hashLib.h**

hookLib

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

ROUTINES

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

DESCRIPTION

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

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

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

NOTE

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

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

INCLUDE FILES **hookLib.h**

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

inflateLib

NAME **inflateLib** – inflate code using public domain zlib functions

ROUTINES **inflate( )** – inflate compressed code

DESCRIPTION This library is used to inflate a compressed data stream, primarily for boot ROM decompression. Compressed boot ROMs contain a compressed executable in the data segment between the symbols **binArrayStart** and **binArrayEnd** (the compressed data is generated by **deflate( )** and **binToAsm**). The boot ROM startup code (in **target/src/config/all/bootInit.c**) calls **inflate( )** to decompress the executable and then jump to it.

This library is based on the public domain zlib code, which has been modified by Wind River Systems. For more information, see the zlib home page at **http://www.gzip.org/zlib/**.

OVERVIEW OF THE COMPRESSION/DECOMPRESSION

1. Compression algorithm (deflate)

The deflation algorithm used by zlib (also zip and gzip) is a variation of LZ77 (Lempel-Ziv 1977, see reference below). It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. (In this description, **string** must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. The blocks can have any size (except that the compressed data for one block must fit in available memory). A block is terminated when **deflate( )** determines that it would be useful to start another block with fresh trees. (This is somewhat similar to the behavior of LZW-based _compress_.)

Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option (level parameter of deflateInit). So **deflate( )** does not always find the longest possible match but generally finds a match which is long enough.

deflate( ) also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, **deflate( )** searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards. Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a runtime parameter. If the current match is long enough, **deflate( )** reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, **deflate( )** attempts a complete second search even if the first match is already long enough.

The lazy match evaluation is not performed for the fastest compression modes (level parameter 1 to 3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

2. Decompression algorithm (zinflate)

The real question is, given a Huffman tree, how to decode fast. The most important realization is that shorter codes are much more common than longer codes, so pay attention to decoding the short codes fast, and let the long codes take longer to decode.

zinflate( ) sets up a first level table that covers some number of bits of input less than the length of longest code. It gets that many bits from the stream, and looks it up in the table. The table will tell if the next code is that many bits or less and how many, and if it is, it will tell the value, else it will point to the next level table for which **zinflate( )** grabs more bits and tries to decode a longer code.

How many bits to make the first lookup is a tradeoff between the time it takes to decode and the time it takes to build the table. If building the table took no time (and if you had infinite memory), then there would only be a first level table to cover all the way to the longest code. However, building the table ends up taking a lot longer for more bits since short codes are replicated many times in such a table. What **zinflate( )** does is simply to make the number of bits in the first table a variable, and set it for the maximum speed.

zinflate( ) sends new trees relatively often, so it is possibly set for a smaller first level table than an application that has only one tree for all the data. For zinflate, which has 286 possible codes for the literal/length tree, the size of the first table is nine bits. Also the distance trees have 30 possible values, and the size of the first table is six bits. Note that for each of those cases, the table ended up one bit longer than the **average** code length, i.e. the code length of an approximately flat code which would be a little more than eight bits for 286 symbols and a little less than five bits for 30 symbols. It would be interesting to see if optimizing the first level table for other applications gave values within a bit or two of the flat code size.

Jean-loup Gailly Mark Adler gzip@prep.ai.mit.edu madler@alumni.caltech.edu

References:

[LZ77] Ziv J., Lempel A., `A Universal Algorithm for Sequential Data Compression,' IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

DEFLATE Compressed Data Format Specification available in ftp://ds.internic.net/rfc/rfc1951.txt

MORE INTERNAL DETAILS

Huffman code decoding is performed using a multi-level table lookup. The fastest way to decode is to simply build a lookup table whose size is determined by the longest code. However, the time it takes to build this table can also be a factor if the data being decoded is not very long. The most common codes are necessarily the shortest codes, so those codes dominate the decoding time, and hence the speed. The idea is you can have a shorter table that decodes the shorter, more probable codes, and then point to subsidiary tables for the longer codes. The time it costs to decode the longer codes is then traded against the time it takes to make longer tables.

This results of this trade are in the variables lbits and dbits below. lbits is the number of bits the first level table for literal/ length codes can decode in one step, and dbits is the same thing for the distance codes. Subsequent tables are also less than or equal to those sizes. These values may be adjusted either when all of the codes are shorter than that, in which case the longest code length in bits is used, or when the shortest code is *longer* than the requested table size, in which case the length of the shortest code in bits is used.

There are two different values for the two tables, since they code a different number of possibilities each. The literal/length table codes 286 possible values, or in a flat code, a little over eight bits. The distance table codes 30 possible values, or a little less than five bits, flat. The optimum values for speed end up being about one bit more than those, so lbits is 8+1 and dbits is 5+1. The optimum values may differ though from machine to machine, and possibly even between compilers. Your mileage may vary.

Notes beyond the 1.93a **appnote.txt**:

1. Distance pointers never point before the beginning of the output stream.
2. Distance pointers can point back across blocks, up to 32k away.

3. There is an implied maximum of 7 bits for the bit length table and 15 bits for the actual data.
4. If only one code exists, then it is encoded using one bit. (Zero would be more efficient, but perhaps a little confusing.) If two codes exist, they are coded using one bit each (0 and 1).
5. There is no way of sending zero distance codes--a dummy must be sent if there are none. (History: a pre 2.0 version of PKZIP would store blocks with no distance codes, but this was discovered to be too harsh a criterion.) Valid only for 1.93a. 2.04c does allow zero distance codes, which is sent as one code of zero bits in length.
6. There are up to 286 literal/length codes. Code 256 represents the end-of-block. Note however that the static length tree defines 288 codes just to fill out the Huffman codes. Codes 286 and 287 cannot be used though, since there is no length base or extra bits defined for them. Similarily, there are up to 30 distance codes. However, static trees define 32 codes (all 5 bits) to fill out the Huffman codes, but the last two had better not show up in the data.
7. Unzip can check dynamic Huffman blocks for complete code sets. The exception is that a single code would not be complete (see #4).
8. The five bits following the block type is really the number of literal codes sent minus 257.
9. Length codes 8,16,16 are interpreted as 13 length codes of 8 bits (1+6+6). Therefore, to output three times the length, you output three codes (1+1+1), whereas to output four times the same length, you only need two codes (1+3). Hmm.
10. In the tree reconstruction algorithm, Code = Code + Increment only if BitLength(i) is not zero. (Pretty obvious.)
11. Correction: 4 Bits: # of Bit Length codes - 4 (4 - 19)
12. Note: length code 284 can represent 227-258, but length code 285 really is 258. The last length deserves its own, short code since it gets used a lot in very redundant files. The length 258 is special since 258 - 3 (the min match length) is 255.
13. The literal/length and distance code bit lengths are read as a single stream of lengths. It is possible (and advantageous) for a repeat code (16, 17, or 18) to go across the boundary between the two sets of lengths.

INCLUDE FILES none

ioLib

NAME **ioLib** – I/O interface library

ROUTINES

lseek( ) – set a file read/write pointer
ioDefPathSet( ) – vxWorks compatible ioDefPathSet (chdir)
ioDefPathGet( ) – get the current default path (VxWorks)
getwd( ) – get the current default path
isatty( ) – return whether the underlying driver is a *tty* device
open( ) – open a file
creat( ) – create a file
remove( ) – remove a file (ANSI) (syscall)
write( ) – write bytes to a file
close( ) – close a file
read( ) – read bytes from a file or device
ioctl( ) – perform an I/O control function
select( ) – pend on a set of file descriptors (syscall)
chdir( ) – change working directory (syscall)
_getcwd( ) – get pathname of current working directory (syscall)
getcwd( ) – get pathname of current working directory
dup( ) – duplicate a file descriptor (syscall)
dup2( ) – duplicate a file descriptor as a specified *fd* number (syscall)
pipeDevCreate( ) – create a named pipe device (syscall)
pipeDevDelete( ) – delete a named pipe device (syscall)
getprlimit( ) – get process resource limits (syscall)
setprlimit( ) – set process resource limits (syscall)
rtpIoTableSizeGet( ) – get *fd* table size for given RTP
rtpIoTableSizeSet( ) – set *fd* table size for given RTP

DESCRIPTION

This library contains the interface to the basic I/O system. It includes:

- Interfaces to several file system functions, including **rename( )** and **lseek( )**.
- Routines to set and get the current working directory.

FILE DESCRIPTORS

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

Three file descriptors are reserved and have special meanings:

0 (**STD_IN**)	standard input
1 (**STD_OUT**)	standard output
2 (**STD_ERR**)	standard error output

CONFIGURATION

This library requires the **INCLUDE_PIPES** component to be configured into the kernel; *errno* will be set to **ENOSYS** if this component is not present.

INCLUDE FILES

ioLib.h, **pipeDrv.h**

SEE ALSO

iosLib, ansiStdio, the VxWorks programmer guides

libc

NAME **libc** – user-side C library routines

ROUTINES

DESCRIPTION This file is solely for information. It does not hold routines. The C library is made of many individual files, each one of them bringing a function traditionally considered as part of the "C library".

Most of the documentation for the user-side C library is provided in the online help of Workbench: Wind River Documentation > References > Standard C and C++ Libraries > Dinkum C++ Library Reference Manual

A few routines are documented separately and appear individually in the manual: **setenv( )**, **unsetenv( )**, **time( )**, **strtok_r( )**

INCLUDE FILES none

loginLib

NAME **loginLib** – user login/password subroutine library

ROUTINES **loginUserVerify( )** – verify a user name and password in the login table

DESCRIPTION This library provides a routine to validate a login/password pair. The login/password pair is looked up within the login user table stored in the kernel and managed using the kernel **loginLib** facility.

Support for login/password validation is included in the system via the **INCLUDE_LOGIN** VxWorks component.

INCLUDE FILES **loginLib.h**

SEE ALSO **loginLib**

lstLib

NAME **lstLib** – doubly linked list subroutine library

ROUTINES

lstInit( ) – initialize a list descriptor
lstAdd( ) – add a node to the end of a list
lstConcat( ) – concatenate two lists
lstCount( ) – report the number of nodes in a list
lstDelete( ) – delete a specified node from a list
lstExtract( ) – extract a sublist from a list
lstFirst( ) – find first node in list
lstGet( ) – delete and return the first node from a list
lstInsert( ) – insert a node in a list after a specified node
lstLast( ) – find the last node in a list
lstNext( ) – find the next node in a list
lstNth( ) – find the Nth node in a list
lstPrevious( ) – find the previous node in a list
lstNStep( ) – find a list node *nStep* steps away from a specified node
lstFind( ) – find a node in a list
lstFree( ) – free up a list

DESCRIPTION

This subroutine library supports the creation and maintenance of a doubly linked list. The user supplies a list descriptor (type LIST) that will contain pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list can be any user-defined structure, but they must reserve space for two pointers as their first elements. Both the forward and backward chains are terminated with a **NULL** pointer.

The linked-list library simply manipulates the linked-list data structures; no kernel functions are invoked. In particular, linked lists by themselves provide no task synchronization or mutual exclusion. If multiple tasks will access a single linked list, that list must be guarded with some mutual-exclusion mechanism (e.g., a mutual-exclusion semaphore).

NON-EMPTY LIST

```
---------             --------         --------
| head--------------->| next--------->| next---------
|       |             |      |        |      |      |
|       |      ------- prev |<--------- prev |      |
|       |      |      |      |        |      |      |
| tail------   |      | ...  |   ----->| ...  |      |
|       |  |   v                 |                  v
|count=2|  |  -----              |                -----
---------  |   ---               |                 ---
           |    -                |                  -
           |                     |
           -----------------------
```

EMPTY LIST

```
 -----------
     |  head------------------
     |       |               |
     |  tail----------       |
     |       |       |       v
     | count=0 |   -----   -----
    -----------     ---     ---
```

– –

INCLUDE FILES **lstLib.h**

memEdrLib

NAME **memEdrLib** – memory manager error detection and reporting library

ROUTINES **memEdrFreeQueueFlush( )** – flush the free queue
memEdrBlockMark( ) – mark or unmark selected blocks

DESCRIPTION This library provides a runtime error detection and debugging tool for memory manager libraries (**memPartLib** and **memLib**). It operates by maintaining a database of blocks allocated, freed and reallocated by the memory manager and by validating memory manager operations using the database.

CONFIGURATION In RTPs, this library can be enabled by symbolically referencing the global *memEdrEnable*. For example, this can be accomplished by using the following linker option: -Wl,-umemEdrEnable. Alternatively, in the application's source code insert:

```
extern int memEdrEnable;
memEdrEnable = TRUE;
```

This library can only be used with applications statically linked with **libc.a**; it cannot be used with applications that are dynamically linked with libc.so.

The following environment variables can be set to alter library configuration on a per process basis:

MEDR_EXTENDED_ENABLE
Set to **TRUE** to enable logging trace information for each allocated block. Defaul setting is **FALSE**.

MEDR_FILL_FREE_ENABLE
Set to **TRUE** to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. Default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN
Lenght of the free queue. Queuing is disabled when this parameter is 0. Default setting is 64.

MEDR_BLOCK_GUARD_ENABLE
Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, underruns, and some heap memory corruption, but results in a per-block allocation overhead of 16 bytes. Default setting is **FALSE**.

MEDR_POOL_SIZE
Set the size of the memory pool used to maintain the memory block database. Default setting is 1MBytes in the kernel, and 64k in RTPs. The database uses 32 bytes per memory block without extended information (call stack trace) enabled, and 64 bytes per block with extended information enabled.

When this library is enabled, the following types of memory manager errors are detected:

- allocating already allocated memory (possible heap corruption)
- allocating with invalid memory partition ID
- freeing a dangling pointer
- freeing non-allocated memory
- freeing a partial block
- freeing global memory
- freeing with invalid partition ID

The errors are logged via the ED&R facility, which should to be included in the kernel configuration. The logs can be viewed with the ED&R show routines and show commands.

FREE QUEUE AND FREE PATTERN

Freed and reallocated blocks are stored in a queue. The queue allows detection of stall pointer dereferencing in freed and re-allocated blocks. The length of the queue is set by **MEDR_FREE_QUEUE_LEN**.

When the **MEDR_FILL_FREE_ENABLE** option is enabled, queued blocks are filled with a special pattern. When the block is removed from the queue, the pattern is matched to detect memory write operations with stale pointer.

When a partition has insufficient memory to satisfy an allocation, the free queue is automatically flushed for that partition. This way the queueing does not cause allocations to fail with insufficient memory while there are blocks in the free queue.

COMPILER INSTRUMENTATION

Code compiled by the Wind River Compiler with RTEC instrumentation enabled (-Xrtc=*code* option) provides automatic pointer reference and pointer arithmetic validation.

For user applications, there is no additional configuration step needed. Whenever any part of the application is built with the -Xrtc compiler option, the supporting user library modules are automatically linked and initialized. The kernel configuration that is used to run such applications should have the ED&R logging facility enabled.

The errors are logged via the ED&R facility, which should to be included in the kernel configuration. The logs can be viewed with the ED&R show routines and show commands.

For more information about the RTEC compiler coption consult the Wind River Compiler documentation.

Note: the stack overflow check option (-Xrtc=0x04) is not supported with this library. Code executed in ISR or kernel context is excluded from compiler instrumentation checks.

CAVEATS

Realloc does not attempt to resize a block. Instead, it will always allocate a new block and enqueue the old block into the free queue. This method enables detection of invalid references to reallocated blocks.

Realloc with size 0 will return a pointer to a block of size 0. This feature coupled with compiler pointer validation instrumentation aids in detecting dereferencing pointers obtained by realloc with size 0.

In order to aid detection of unintended free and realloc operation on invalid pointers, memory partitions should not be created in a task's stack when this library is enabled. Although it is possible to create such memory partitions, it is not a recommended practice; this library will flag it as an error when an allocated block is within a tasks's own stack.

Memory partition information is recorded in the database for each partition created. This information is kept even after the memory partition is deleted, so that unintended operations with a deleted partition can be detected.

INCLUDE FILES

none

SEE ALSO

memEdrShow, **memEdrRtpShow**, **edrLib**, **memLib**, **memPartLib**

memLib

NAME

memLib – user heap manager

ROUTINES

memAddToPool( ) – add memory to the RTP memory partition
malloc( ) – allocate a block of memory from the RTP heap (ANSI)
free( ) – free a block of memory from the RTP heap (ANSI)
memalign( ) – allocate aligned memory from the RTP heap
valloc( ) – allocate memory on a page boundary from the RTP heap
memOptionsSet( ) – set the options for the RTP heap
memOptionsGet( ) – get the options for the RTP heap
calloc( ) – allocate space for an array from the RTP heap (ANSI)
realloc( ) – reallocate a block of memory from the RTP heap (ANSI)
cfree( ) – free a block of memory from the RTP heap
memFindMax( ) – find the largest free block in the RTP heap
memInfoGet( ) – get heap information

DESCRIPTION

This library provides the API for allocating and freeing blocks of memory of arbitrary size from an RTP's heap. This library implements an RTP heap as a dedicated memory partition. One private heap is created automatically for every RTP.

The library provides ANSI allocation routines and enhanced memory management features, including error handling, aligned allocation. Most of the **memLib** routines are simple wrapper to the memory partition management functions which implement the actual memory management functionalities. For more information about the memory partition management facility, see the reference entry for **memPartLib**.

HEAP OPTIONS

Various options can be selected for the current heap using **memOptionsSet( )**. For the actual options that are supported, refer to **memPartLib** and **memPartOptionsSet( )**.

ENVIRONMENT VARIABLES

The heap can be controlled with the help of three environment variables:

HEAP_INITIAL_SIZE
If **HEAP_INITIAL_SIZE** is set to a positive, non-zero value, then it specifies the initial size of the RTP heap. However, the minimum initial heap size for an RTP is always at least as large as the value set by the system-wide RTP component configuration parameter **RTP_HEAP_INIT_SIZE**. The actual initial size is round up to a multiple of the virtual memory page size.

HEAP_INCR_SIZE
If **HEAP_INCR_SIZE** is set to a positive, non-zero value, then the RTP heap is authorized grow the RTP heap space by an amount multiple of this increment value any time it runs out of free space. The actual increment is always rounded up to a multiple of the virtual memory page size. If **HEAP_INCR_SIZE** is set to zero, then heap growth is disabled. If it's not set or is set to a negative number, then the heap grows with multiples of the virtual memory page size.

HEAP_MAX_SIZE
If **HEAP_MAX_SIZE** is set, the RTP heap will not grow beyond this value. If it's not set, or it's set to zero or a negative number, then there is no limit for RTP heap growth. **HEAP_MAX_SIZE** however does not limit the RTP's initial heap size; it is only checked during heap auto-growth.

HEAP_OPTIONS
If **HEAP_OPTIONS** is set, then that value would be used for the default options for the RTP heap. If it's not set, then the value obtained by the aux vector **AT_WINDHEAPOPT** will be used as the default options for the RTP heap.

INCLUDE FILES

memLib.h, **stdlib.h**

SEE ALSO

memPartLib

memPartLib

NAME

memPartLib – user level memory partition manager

ROUTINES

memPartCreate( ) – create a memory partition
memPartDelete( ) – delete a partition and free associated memory
memPartAddToPool( ) – add memory to a memory partition
memPartAlignedAlloc( ) – allocate aligned memory from a partition
memPartAlloc( ) – allocate a block of memory from a partition
memPartFree( ) – free a block of memory in a partition
memPartRealloc( ) – reallocate a block of memory in a specified partition
memPartOptionsGet( ) – get the options of a memory partition
memPartOptionsSet( ) – set the debug options for a memory partition
memPartFindMax( ) – find the size of the largest free block
memPartInfoGet( ) – get partition information

DESCRIPTION

This user library provides core facilities for managing the allocation of memory blocks from ranges of memory called memory partitions. The library was designed to provide full-featured memory management functionality. This library comprises a general facility for the creation and management of memory partitions, and for the allocation and deallocation of blocks from those partitions.

The allocation of memory, using **memPartAlloc( )** for a specific memory partition, is performed with a best-fit algorithm. Adjacent blocks of memory are coalesced when they are freed with **memPartFree( )**. There is also a routine provided for allocating memory aligned to a specified boundary from a specific memory partition, **memPartAlignedAlloc( )**.

Memory partitions are always local to a process. This means a partition created by an RTP cannot be shared by other RTPs, even if the partition memory is in a shared data region (SD). Only the RTP that created the partition is allowed to allocate from it and free into it. However, buffers allocated from a partition created from SD memory can be shared among processes that opened the same SD.

Various debug options can be selected for each partition using **memPartOptionsSet( )** and **memOptionsSet( )**. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. There are four error-handling options that can be individually selected:

MEM_ALLOC_ERROR_EDR_FATAL_FLAG
: Inject a fatal ED&R event when there is an error in allocating memory. This option takes precedence over the **MEM_ALLOC_ERROR_EDR_WARN_FLAG** and **MEM_ALLOC_ERROR_SUSPEND_FLAG** options.

MEM_ALLOC_ERROR_EDR_WARN_FLAG
: Inject a non-fatal ED&R event when there is an error in allocating memory.

MEM_ALLOC_ERROR_LOG_FLAG
Log a message when there is an error in allocating memory.

MEM_ALLOC_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in allocating memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended). This option has been deprecated (available for backward compatibility only).

MEM_BLOCK_ERROR_EDR_FATAL_FLAG
Inject a fatal ED&R event when there is an error in freeing or reallocating memory. This option takes precedence over the **MEM_BLOCK_ERROR_EDR_WARN_FLAG** and **MEM_BLOCK_ERROR_SUSPEND_FLAG** options.

MEM_BLOCK_ERROR_EDR_WARN_FLAG
Inject a non-fatal ED&R event when there is an error in freeing or reallocating memory.

MEM_BLOCK_ERROR_LOG_FLAG
Log a message when there is an error in freeing memory.

MEM_BLOCK_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in freeing memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended). This option has been deprecated (available for backward compatibility only).

When the following option is specified to check every block freed to the partition, **memPartFree( )** and **free( )** in **memPartLib** run consistency checks of various pointers and values in the header of the block being freed. If this flag is not specified, no check will be performed when memory is freed.

MEM_BLOCK_CHECK
Check each block freed.

Setting any of the **MEM_BLOCK_ERROR_** options automatically sets **MEM_BLOCK_CHECK**.

The default options when a partition is created are:

MEM_ALLOC_ERROR_LOG_FLAG
MEM_ALLOC_ERROR_EDR_WARN_FLAG
MEM_BLOCK_CHECK
MEM_BLOCK_ERROR_LOG_FLAG
MEM_BLOCK_ERROR_EDR_FATAL_FLAG

When setting options for a partition with **memPartOptionsSet( )** or **memOptionsSet( )**, use the logical OR operator between each specified option to construct the *options* parameter. For example:

```
memPartOptionsSet (myPartId, MEM_ALLOC_ERROR_LOG_FLAG |
                             MEM_BLOCK_CHECK |
                             MEM_BLOCK_ERROR_LOG_FLAG);
```

In the case when multiple options are set so that one option takes precedence over the other, then the preceeded options may not have their expected effect. For example, if the

MEM_BLOCK_ERROR_EDR_FATAL_FLAG flag results in a task being stopped by the ED&R fatal policy handler, then the **MEM_BLOCK_ERROR_SUSPEND_FLAG** flag has no effect (a task cannot be stopped and suspended at the same time).

CAVEATS

Architectures have various alignment constraints. To provide optimal performance, **memPartAlloc( )** returns a pointer to a buffer having the appropriate alignment for the architecture in use. The portion of the allocated buffer reserved for system bookkeeping, known as the overhead, may vary depending on the architecture. The following table lists the default alignment and overhead size of free and allocated memory blocks for various architectures.

Architecture	Boundary	Overhead
ARM	4	16
COLDFIRE	4	16
I86	4	16
M68K	4	16
MCORE	8	16
MIPS	16	16
PPC (*)	8-16	16
SH	4	16
SIMLINUX	8	16
SIMNT	8	16
SIMSOLARIS	8	16
SPARC	8	16

(*) On PowerPC, the boundary and allocated block overhead values are 16 bytes for system based on the PPC604 CPU type (including ALTIVEC). For all other PowerPC CPU types (PPC403, PPC405, PPC440, PPC860, PPC603, etc...), the boundary for allocated blocks is 8 bytes.

The partition's free blocks are organized into doubly linked lists. Each list contains only free blocks of the same size. The head of these doubly linked lists are organized in an AVL tree. The memory for the AVL tree's nodes is carved out from the partition itself, whenever new AVL nodes need to be created. This occurs only if the fragmentation of the partition increases; to be more exact, it happens only if a free memory block is created whose size does not have a doubly linked list yet.

INCLUDE FILES

memPartLib.h, **stdlib.h**

SEE ALSO

memLib

mmanLib

NAME

mmanLib – memory management library

ROUTINES

mmap( ) – map pages of memory (syscall)
munmap( ) – unmap pages of memory (syscall)
msync( ) – synchronize a file with a physical storage
mprotect( ) – set protection of memory mapping (syscall)
mlockall( ) – lock all pages used by a process into memory
munlockall( ) – unlock all pages used by a process
mlock( ) – lock specified pages into memory
munlock( ) – unlock specified pages
mprobe( ) – probe memory mapped in process
_mctl( ) – invoke memory control functions (syscall)

DESCRIPTION

This library provides the API for managing memory pages for an RTP. It allows an application to request pages of memory mapped in the RTP's context, unmap previously mapped memory, or change protection attributes of mapped memory pages. It also provides the API for POSIX page locking options, although these APIs currently are mostly no-ops: in VxWorks all mapped pages are memory resident.

Three types of mappings are supported by the **mmap( )** implemented in this library:

Anonymous
The mapping is established directly with the system RAM. Only private mappings are supported. This is the simplest way to extend the address space of a process. This mapping type is always supported with RTPs.

Shared Memory Objects
The file descriptor is obtained with **shm_open( )**. Both shared and private mappings are supported. This mapping type is available when the **INCLUDE_POSIX_SHM** and the **INLUDE_POSIX_MAPPED_FILES** components are included in the kernel configuration.

Memory Mapped Files
The file descriptor is obtained by opening a regular file in a POSIX-conformant file system. Both shared and private mappings are supported. This mapping type is available when the **INLUDE_POSIX_MAPPED_FILES** is included in the kernel configuration.

MEMORY RESIDENT MAPPINGS

Mappings established with this library are always memory-resident. Demand paging and copy-on write are not performed. This ensures deterministic memory access for mapped files, but it also means that physical memory is continuously associated to mappings, until unmapped. Also, this implicitly means that all pages are always locked in memory, therefore the memory locking APIs are no-ops.

FILE SYNCHRONIZATION

For memory mapped files there is no automatic synchronization, and there is no unified buffering for **mmap( )** and the file system. This means the application must use **msync( )** to synchronize a mapped image with the file's permanent storage. The only exception is when memory is unmapped explicitly with **munmap( )**, or unmapped implicitly when the process exits; in that case the synchronization is performed automatically during the unmapping process.

EXAMPLE

The following example shows a typical usage for memory mapped regular files. This example assumes /tmp has been mounted using a POSIX conformant file system.

```
/*
 * Create a new data file.
 */

#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

int main ()
    {
    int   fd;
    int   ix;
    int * pData;

    /* create a new file */

    fd = open("/tmp/datafile", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1)
        exit (1);

    /* set file size */

    if (ftruncate (fd, 0x1000) == -1)
        exit (1);

    /* Map file in the address space of the process */

    pData = (int *) mmap (0, 0x1000,  PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);

    if (pData == (int *) MAP_FAILED)
        exit (1);

    /* close the file descriptor; the mapping is not impacted by this */

    close (fd);

    /* The mapped image can now be written via the pData pointer */

    for (ix = 0; ix < 25; ix++)
        *(pData + ix) = ix;

    /* synchronize file */

    if (msync (pData, 0x1000, MS_SYNC) == -1)
```

```
        exit (1);

    /* when the process exits, the object is automatically unmapped */

    exit (0);
    }
```

For another usage example with shared memory objects, see the **shmLib** library guide.

INCLUDE FILES **sys/mman.h**

SEE ALSO POSIX 1003.1, **shmLib**

mqPxLib

NAME **mqPxLib** – user-level message queue library (POSIX)

ROUTINES
mq_open( ) – open a message queue (POSIX)
mq_receive( ) – receive a message from a message queue (POSIX)
mq_timedreceive( ) – receive a message from a message queue with timeout (POSIX)
mq_send( ) – send a message to a message queue (POSIX)
mq_timedsend( ) – send a message to a message queue with timeout (POSIX)
mq_close( ) – close a message queue (POSIX)
mq_unlink( ) – remove a message queue (POSIX)
mq_notify( ) – notify a task that a message is available on a queue (POSIX)
mq_setattr( ) – set message queue attributes (POSIX)
mq_getattr( ) – get message queue attributes (POSIX)

DESCRIPTION This library implements the user-level message-queue interface based on the POSIX 1003.1b standard, as an alternative to the VxWorks-specific message queue design in **msgQLib**. The POSIX message queues are accessed through names; each message queue supports multiple sending and receiving tasks.

The message queue interface imposes a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis. The value may not be changed dynamically.

This interface allows a task to be notified asynchronously of the availability of a message on the queue. The purpose of this feature is to let the task perform other functions and yet still be notified that a message has become available on the queue.

MESSAGE QUEUE DESCRIPTOR DELETION

The **mq_close( )** call terminates a message queue descriptor and deallocates any associated memory. When deleting message queue descriptors, take care to avoid interfering with other tasks that are using the same descriptor. Tasks should only close message queue descriptors that the same task has opened successfully.

MESSAGE QUEUE NAME LENGTH

The message queue namespace in VxWorks is not associated with the filesystem. Thus, it is incorrect to use the POSIX variable values **NAME_MAX** and **PATH_MAX** to specify the maximum length of the message queue name and path. Instead, **_VX_PX_MQ_NAME_MAX** and **_VX_PX_MQ_PATH_MAX** are defined for the corresponding length limits for message queue names. The semantic of the **_VX_PX_MQ_PATH_MAX** is the same as for **PATH_MAX**, and the semantic of the **_VX_PX_MQ_NAME_MAX** is the same as for **NAME_MAX**.

For the same reason, POSIX **pathconf( )** API must not be used with message queue object path.

CONFIGURATION

This library requires the **INCLUDE_POSIX_MQ** component to be configured into the kernel; *errno* may be set to **ENOSYS** if this component is not present.

The **INCLUDE_SIGEVENT** component is required to support the **SIGEV_THREAD** notification type. The **SIGEV_THREAD** notification type also requires POSIX thread support which requires the POSIX Clocks and Pthread Scheduler components (**INCLUDE_POSIX_CLOCKS** and **INCLUDE_POSIX_PTHREAD_SCHEDULER**) to be included in the VxWorks kernel; *errno* may be set to **ENOSYS** if these components have not been configured into the kernel.

INCLUDE FILES

mqueue.h

SEE ALSO

POSIX 1003.1b document, **msgQLib**, the VxWorks programmer guides.

msgQEvLib

NAME

msgQEvLib – VxWorks user events support for message queues

ROUTINES

msgQEvStart( ) – start event notification process for a message queue
msgQEvStop( ) – stop the event notification process for a message queue

DESCRIPTION

This library is an extension to **eventLib**, the events library. Its purpose is to support events for message queues.

The functions in this library are used to control registration of tasks on a message queue. The routine **msgQEvStart( )** registers a task and starts the notification process. The function **msgQEvStop( )** un-registers the task, which stops the notification mechanism.

When a task is registered and a message arrives on the queue, the events specified are sent to that task, on the condition that no other task is pending on that message queue. However, if a **msgQReceive( )** is to be done afterwards to get the message, there is no guarantee that it will still be available.

INCLUDE FILES **msgQEvLib.h**

SEE ALSO **eventLib**, **msgQLib**

msgQInfo

NAME **msgQInfo** – user-level message queue information routines

ROUTINES **msgQInfoGet( )** – get information about a message queue

DESCRIPTION This library provides the routine **msgQInfoGet( )** to extract message queue statistics, such as the task queuing method, messages queued, and receivers blocked.

INCLUDE FILES **msgQLib.h**

SEE ALSO **msgQLib**, **msgQShow** (kernel)

msgQLib

NAME **msgQLib** – user-level message queue library

ROUTINES
msgQOpen( ) – open a message queue
msgQClose( ) – close a named message queue
msgQUnlink( ) – unlink a named message queue
msgQCreate( ) – create and initialize a message queue
msgQDelete( ) – delete a message queue
msgQNumMsgs( ) – get the number of messages queued to a message queue
_msgQOpen( ) – open a message queue (system call)
msgQSend( ) – send a message to a message queue (system call)
msgQReceive( ) – receive a message from a message queue (system call)

DESCRIPTION This library contains routines for creating and using message queues, the primary intertask communication mechanism within a single CPU. Message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out (FIFO) order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

CREATING AND USING MESSAGE QUEUES

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

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

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

TIMEOUTS

Both **msgQSend( )** and **msgQReceive( )** take timeout parameters. When sending a message, if no buffer space is available to queue the message, the timeout specifies how many ticks to wait for space to become available. When receiving a message, the timeout specifies how many ticks to wait if no message is immediately available. The *timeout* parameter can have the special values **NO_WAIT** (0) or **WAIT_FOREVER** (-1). **NO_WAIT** means the routine returns immediately; **WAIT_FOREVER** means the routine never times out.

URGENT MESSAGES

The **msgQSend( )** routine allows the priority of a message to be specified. It can be either **MSG_PRI_NORMAL** (0) or **MSG_PRI_URGENT** (1). Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

VXWORKS EVENTS

If a task has registered with a message queue using **msgQEvStart( )**, events are sent to that task when a message arrives on that message queue, if no other task is pending on the queue.

CONFIGURATION

This library requires the **INCLUDE_MSG_Q** component to be configured into the kernel; *errno* will be set to **ENOSYS** if this component is not present.

INCLUDE FILES

msgQLib.h

SEE ALSO

msgQEvLib, **eventLib**

objLib

NAME **objLib** – VxWorks user object management library

ROUTINES
objDelete( ) – generic object delete/close routine (system call)
objInfoGet( ) – generic object information retrieve routine (system call)
objUnlink( ) – unlink an object (system call)

DESCRIPTION This library provides the interface to the VxWorks user object management facilities.

INCLUDE FILES none

poolLib

NAME **poolLib** – Memory Pool Library

ROUTINES
poolCreate( ) – create a pool
poolDelete( ) – delete a pool
poolBlockAdd( ) – add an item block to the pool
poolUnusedBlocksFree( ) – free blocks that have all items unused
poolItemGet( ) – get next free item from pool and return a pointer to it
poolItemReturn( ) – return an item to the pool
poolIncrementSet( ) – set the increment value used to grow the pool
poolIncrementGet( ) – get the increment value used to grow the pool
poolTotalCount( ) – return total number of items in pool
poolFreeCount( ) – return number of free items in pool

DESCRIPTION This module contains the Memory Pool library. Pools provide a fast and efficient memory management when an aplication uses a large number of identically sized memory items (e.g. structures, objects) by minimizing the number of allocations from a memory partition. The use of pools also reduces possible fragmentation caused by frequent memory allocation and freeing.

A pool is a dynamic set of statically sized memory items. All items in a pool are of the same size, and all are guaranteed a power of two alignment. The size and alignemnt of items are specified at pool creation time. An item can be of arbitrary size, but the actual memory used up by each item is at least 8 bytes, and it is a multiple of the item alignment. The minimum alignment of items is the architecture specific allocation alignment.

Pools are created and expanded using a specified number of items for initial size and another number of items for incremental pool additions. The initial set of items and the incremental pool items are added as one block of memory. Each memory block can be allocated from either the system memory partition (when the partition ID passed to **poolCreate( )** is **NULL**), a user-provided memory partition. A block can be also added to the pool using any memory specified by the user using **poolBlockAdd( )**. For example, if all items in a pool have to be in some specific memory zone, the pool can be created with initial and incremental item count as zero in order to prevent automatic creation of blocks from memory partitions, and explicitly adding blocks with **poolBlockAdd( )** as needed. The memory provided to the pool must be writable. Allocation and free from memory pools are performed using the **poolItemGet( )** and **poolItemReturn( )** routines.

If the pool item increment is specified as zero, the pool will be static, unable to grow dynamically. A static pool is more deterministic.

Pools are intended for use in systems requiring frequent allocating and freeing of memory in statically sized blocks such as used in messaging systems, data- bases, and the like. This pool system is dynamic and grows upon request, eventually allowing a system to achieve a stable state with no further memory requests needed.

INCLUDE FILE **poolLib.h**

SEE ALSO **memPartLib**

posixScLib

NAME **posixScLib** – POSIX message queue and semaphore system call documentation

ROUTINES
pxOpen( ) – open a POSIX semaphore or message queue (syscall)
pxClose( ) – close a reference to a POSIX semaphore or message queue (syscall)
pxUnlink( ) – unlink the name of a POSIX semaphore or message queue (syscall)
pxMqReceive( ) – receive a message from a POSIX message queue (syscall)
pxMqSend( ) – send a message to a POSIX message queue (syscall)
pxSemPost( ) – post a POSIX semaphore (syscall)
pxSemWait( ) – wait for a POSIX semaphore (syscall)
pxCtl( ) – control operations on POSIX semaphores and message queues (syscall)

DESCRIPTION This module contains system call documentation for POSIX message queue and semaphore system calls.

CONFIGURATION This library requires the **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components to be configured into the kernel; *errno* will be set to **ENOSYS** if these components are not present

INCLUDE FILES **pxObjSysCall.h**, **semPxSysCall.h**, **mqPxSysCall.h**

pthreadLib

NAME **pthreadLib** – POSIX 1003.1/1003.13 (PSE52) thread library interfaces

ROUTINES
pthread_sigmask( ) – change and/or examine calling thread's signal mask (POSIX)
pthread_kill( ) – send a signal to a thread (POSIX)
pthread_atfork( ) – register fork handlers (POSIX)
pthread_mutexattr_init( ) – initialize mutex attributes object (POSIX)
pthread_mutexattr_destroy( ) – destroy mutex attributes object (POSIX)
pthread_mutexattr_setprotocol( ) – set protocol attribute in mutex attribute object (POSIX)
pthread_mutexattr_getprotocol( ) – get value of protocol in mutex attributes object (POSIX)
pthread_mutexattr_setprioceiling( ) – set prioceiling attribute in mutex attributes object (POSIX)
pthread_mutexattr_getprioceiling( ) – get the current value of the prioceiling attribute in a mutex attributes object (POSIX)
pthread_mutexattr_settype( ) – set type attribute in mutex attributes object (POSIX)
pthread_mutexattr_gettype( ) – get the current value of the type attribute in a mutex attributes object (POSIX)
pthread_mutex_getprioceiling( ) – get the value of the prioceiling attribute of a mutex (POSIX)
pthread_mutex_setprioceiling( ) – dynamically set the prioceiling attribute of a mutex (POSIX)
pthread_mutex_init( ) – initialize mutex from attributes object (POSIX)
pthread_mutex_destroy( ) – destroy a mutex (POSIX)
pthread_mutex_lock( ) – lock a mutex (POSIX)
pthread_mutex_timedlock( ) – lock a mutex with timeout (POSIX)
pthread_mutex_trylock( ) – lock mutex if it is available (POSIX)
pthread_mutex_unlock( ) – unlock a mutex (POSIX)
pthread_condattr_init( ) – initialize a condition attribute object (POSIX)
pthread_condattr_destroy( ) – destroy a condition attributes object (POSIX)
pthread_cond_init( ) – initialize condition variable (POSIX)
pthread_cond_destroy( ) – destroy a condition variable (POSIX)
pthread_cond_signal( ) – unblock a thread waiting on a condition (POSIX)
pthread_cond_broadcast( ) – unblock all threads waiting on a condition (POSIX)
pthread_cond_wait( ) – wait for a condition variable (POSIX)
pthread_cond_timedwait( ) – wait for a condition variable with a timeout (POSIX)
pthread_attr_setscope( ) – set contention scope for thread attributes (POSIX)
pthread_attr_getscope( ) – get contention scope from thread attributes (POSIX)
pthread_attr_setinheritsched( ) – set inheritsched attribute in thread attribute object (POSIX)
pthread_attr_getinheritsched( ) – get current value if inheritsched attribute in thread attributes object (POSIX)
pthread_attr_setschedpolicy( ) – set schedpolicy attribute in thread attributes object (POSIX)

pthread_attr_getschedpolicy( ) – get schedpolicy attribute from thread attributes object (POSIX)
pthread_attr_setschedparam( ) – set schedparam attribute in thread attributes object (POSIX)
pthread_attr_getschedparam( ) – get value of schedparam attribute from thread attributes object (POSIX)
pthread_getschedparam( ) – get value of schedparam attribute from a thread (POSIX)
pthread_setschedparam( ) – dynamically set schedparam attribute for a thread (POSIX)
pthread_setschedprio( ) – dynamically set priority attribute for a thread (POSIX)
pthread_attr_init( ) – initialize thread attributes object (POSIX)
pthread_attr_destroy( ) – destroy a thread attributes object (POSIX)
pthread_attr_setopt( ) – set options in thread attribute object
pthread_attr_getopt( ) – get options from thread attribute object
pthread_attr_setname( ) – set name in thread attribute object
pthread_attr_getname( ) – get name of thread attribute object
pthread_attr_setstacksize( ) – set stack size in thread attributes object (POSIX)
pthread_attr_getstacksize( ) – get stack value of stacksize attribute from thread attributes object (POSIX)
pthread_attr_setstackaddr( ) – set stackaddr attribute in thread attributes object (POSIX)
pthread_attr_getstackaddr( ) – get value of stackaddr attribute from thread attributes object (POSIX)
pthread_attr_setstack( ) – set stack attributes in thread attributes object (POSIX)
pthread_attr_getstack( ) – get stack attributes from thread attributes object (POSIX)
pthread_attr_setguardsize( ) – set the thread guard size (POSIX)
pthread_attr_getguardsize( ) – get the thread guard size (POSIX)
pthread_attr_setdetachstate( ) – set detachstate attribute in thread attributes object (POSIX)
pthread_attr_getdetachstate( ) – get value of detachstate attribute from thread attributes object (POSIX)
pthread_create( ) – create a thread (POSIX)
pthread_detach( ) – dynamically detach a thread (POSIX)
pthread_join( ) – wait for a thread to terminate (POSIX)
pthread_exit( ) – terminate a thread (POSIX)
pthread_equal( ) – compare thread IDs (POSIX)
pthread_self( ) – get the calling thread's ID (POSIX)
pthread_once( ) – dynamic package initialization (POSIX)
pthread_key_create( ) – create a thread specific data key (POSIX)
pthread_setspecific( ) – set thread specific data (POSIX)
pthread_getspecific( ) – get thread specific data (POSIX)
pthread_key_delete( ) – delete a thread specific data key (POSIX)
pthread_cancel( ) – cancel execution of a thread (POSIX)
pthread_setcancelstate( ) – set cancellation state for calling thread (POSIX)
pthread_setcanceltype( ) – set cancellation type for calling thread (POSIX)
pthread_testcancel( ) – create a cancellation point in the calling thread (POSIX)
pthread_cleanup_push( ) – pushes a routine onto the cleanup stack (POSIX)
pthread_cleanup_pop( ) – pop a cleanup routine off the top of the stack (POSIX)

pthread_setconcurrency( ) – set the level of concurrency (POSIX)
pthread_getconcurrency( ) – get the level of concurrency (POSIX)

DESCRIPTION

This library provides an implementation of POSIX 1003.1 threads for VxWorks applications (Real Time Processes) in agreement with the PSE52 profile of the IEEE 1003.13 standard. This provides an increased level of compatibility between VxWorks applications and those written for other operating systems that support the POSIX threads model (often called *pthreads*).

VxWorks is primarily a task based operating system, rather than one implementing the process model in the POSIX sense. However VxWorks also introduces the concept of Real Time Process (RTP) which, although a non-schedulable entity, has many of the traditional aspects of a process model. As a result of this, there are a few restrictions in the implementation, but in general, since tasks are roughly equivalent to threads, the *pthreads* support maps well onto VxWorks. The restrictions are explained in more detail in the following paragraphs.

CONFIGURATION

pThreads support in RTP also requires the POSIX Clocks and Pthread Scheduler components (**INCLUDE_POSIX_CLOCKS** and **INCLUDE_POSIX_PTHREAD_SCHEDULER**) to be included in the VxWorks kernel; *errno* may be set to **ENOSYS** if these components have not been configured into the kernel.

THREADS

A thread is essentially a VxWorks task, with some additional characteristics. The first is detachability, where the creator of a thread can optionally block until the thread exits. The second is cancelability, where one task or thread can cause a thread to exit, possibly calling cleanup handlers. The next is private data, where data private to a thread is created, accessed and deleted via keys. Each thread has a unique ID. A thread's ID is different than it's VxWorks task ID.

It is recommended to use the POSIX thread API only via POSIX threads, not via native VxWorks tasks. Since pthreads are not created by default in VxWorks the **pthread_create( )** API can be safely used by a native VxWorks task in order to create the first POSIX thread. If a native VxWorks task must use more pthread API it is recommended to give this task a pthread persona by calling **pthread_self( )** first. Note that this is not required for the RTP's initial task which already has a pthread persona when POSIX threads are used in the RTP.

MUTEXES

Included with the POSIX threads facility is a mutual exclusion facility, or *mutex*. These are functionally similar to the VxWorks mutex semaphores (see **semMLib** for more detail), and in fact are implemented using a VxWorks user-level mutex semaphore. The advantage they offer, like all of the POSIX libraries, is the ability to run software designed for POSIX platforms under VxWorks.

There are three types of locking protocols available: **PTHREAD_PRIO_NONE**, **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**. **PTHREAD_PRIO_INHERIT** maps to a semaphore created with **SEM_Q_PRIORITY** and **SEM_INVERSION_SAFE** set (see **semMCreate** for more detail). A thread locking a mutex created with its protocol attribute set to **PTHREAD_PRIO_PROTECT** has its priority elevated to that of of the prioceiling attribute of the mutex. When the mutex is unlocked, the priority of the calling thread is restored to its previous value. Both protocols aim at solving the priority inversion problem where a lower priority thread can unduly delay a higher priority thread requiring the resource blocked by the lower priority thread. The **PTHREAD_PRIO_INHERIT** protocol can be more efficient since it elevates the priority of a thread only when needed. The **PTHREAD_PRIO_PROTECT** protocol gives more control over the priority change at the cost of systematically elevating the thread's priority as well as preventing threads to use a mutex which priority ceiling is lower than the thread's priority. In contrast the **PTHREAD_PRIO_NONE** protocol, which is the default, does not affect the priority and scheduling of the thread that owns the mutex.

POSIX defines four types of mutex. Valid mutex types are: **PTHREAD_MUTEX_NORMAL** - this type of mutex does not provide deadlock detection. Attempting to relock a mutex causes deadlock. Attempting to unlock a mutex that is owned by another thread or unlock a mutex that is not locked returns error. **PTHREAD_MUTEX_ERRORCHECK** - this type of mutex provides error checking. Attempting to relock a mutex will return error. Attempting to unlock a mutex that is owned by another thread or unlock a mutex that is not locked returns error. **PTHREAD_MUTEX_RECURSIVE** - this type of mutex allows relocking by a thread. Multiple locks of the mutex will require the same number of unlocks to release the mutex. Attempting to unlock a mutex that is owned by another thread or unlock a mutex that is not locked returns error. **PTHREAD_MUTEX_DEFAULT** - set to **PTHREAD_MUTEX_NORMAL** in VxWorks implementation. The default type of mutex is **PTHREAD_MUTEX_DEFAULT**.

CONDITION VARIABLES

Condition variables are another synchronization mechanism that is included in the POSIX threads library. A condition variable allows threads to block until some condition is met. There are really only two basic operations that a condition variable can be involved in: waiting and signaling. Condition variables are always associated with a mutex.

A thread can wait for a condition to become true by taking the mutex and then calling **pthread_cond_wait( )**. That function will release the mutex and wait for the condition to be signaled by another thread. When the condition is signaled, the function will re-acquire the mutex and return to the caller.

Condition variable support two types of signaling: single thread wake-up using **pthread_cond_signal( )**, and multiple thread wake-up using **pthread_cond_broadcast( )**. The latter of these will unblock all threads that were waiting on the specified condition variable.

It should be noted that condition variable signals are not related to POSIX signals. In fact, they are implemented using VxWorks user-level semaphores.

STACK GUARD AREA

Stack overflow protection is provided by setting a non null size for the stack guard area (see **pthread_attr_setguardsize( )**). This protection is limited to the execution stack. If a more extended protection is required it can be obtained via VxWorks' native global stack protection mechanism (guard zones). See the documentation about **taskInitExcStk( )** and the **INCLUDE_PROTECT_TASK_STACK** parameter.

RESOURCE COMPETITION

All tasks, and therefore all POSIX threads, compete for CPU time together. For that reason the contention scope thread attribute is always **PTHREAD_SCOPE_SYSTEM** even when threads run in a real-time process.

NO VXWORKS EQUIVALENT

At the moment there is no notion of sharing of locks (mutexes) and condition variables between RTPs. As a result, the POSIX symbol **_POSIX_THREAD_PROCESS_SHARED** is defined with the value -1 in this implementation, and the routines **pthread_condattr_getpshared( )**, **pthread_condattr_setpshared( )**, **pthread_mutexattr_getpshared( )** are not implemented.

Also, since VxWorks' Real Time Process concept is not using the **fork( )**/**exec( )** model, **pthread_atfork( )** always returns **ERROR**. This routine is provided to satisfy linkage requirements of applications but is not meant to be used.

SCHEDULING

POSIX threads can be scheduled using different policies: **SCHED_FIFO**, **SCHED_RR** and **SCHED_OTHER**. Unlike VxWorks tasks, which are submitted to the system's global scheduling policy, the POSIX scheduling policy is an attribute of a thread and can be assigned and changed on a per-thread basis.

SCHED_FIFO is a preemptive priority scheduling policy. For a given priority level threads scheduled with this policy are handled as peers of the VxWorks tasks at the same level. Remember that POSIX thread priority scheme is the reverse of the VxWorks task priority scheme.

SCHED_RR is a per-priority round-robin scheduling policy. For a given priority level all threads scheduled with this policy are given the same time of execution before giving up the CPU.

SCHED_OTHER corresponds to the active VxWorks native scheduling policy, i.e. either preemptive priority or round-robin. Threads scheduled with this policy are submitted to the system's global scheduling policy, exactly like VxWorks tasks.

SCHED_SPORADIC is identical to the **SCHED_FIFO** policy with some additional conditions that cause the thread's assigned priority to be switched between the sched_priority and sched_ss_low_priority. The conditions includes the thread execution time, execution capacity, execution replenishment period, and the number of the replenishment events. The **SCHED_SPORADIC** is configured to VxWorks only when **INCLUDE_PX_SCHED_SPORADIC_POLICY** component is included. Current implementation uses system periodic timer for time accouting, and does not allow dynamically changing the scheduling policies of threads to the **SCHED_SPORADIC** policy.

The default scheduling policy applied when a thread is created is inherited from its parent, whether VxWorks task or POSIX thread. If a different scheduling policy is to be used, a thread attribute object specifying the scheduling policy and priority must be created and passed to the **pthread_create( )** API. Note that these attributes will take effect only if the attribute object also specifies the explicit scheduling mode (**PTHREAD_EXPLICIT_SCHED**) set via the **pthread_attr_setinheritsched( )** API.

CREATION AND CANCELLATION

Each time a thread is created, the *pthreads* library allocates resources on behalf of it. Each time a VxWorks task (i.e. one not created by the **pthread_create( )** function) uses a POSIX threads feature such as thread private data or pushes a cleanup handler, the *pthreads* library creates resources on behalf of that task as well.

Asynchronous thread cancellation is accomplished by way of a signal. A special signal, SIGCNCL, has been set aside in this version of VxWorks for this purpose. Applications should take care not to block or handle SIGCNCL.

Current cancellation points in system and library calls:

Libraries	**cancellation points**
aioPxLib	aio_suspend
clockLib	clock_nanosleep
ioLib	creat, open, read, write, close, fsync, fdatasync, fcntl
mqPxLib	mq_receive, mq_send, mq_timedreceive, mq_timedsend
mmanLib	msync
pthreadLib	pthread_cond_timedwait, pthread_cond_wait, pthread_join, pthread_testcancel
semPxLib	sem_timedwait, sem_wait
sigLib	pause, sigsuspend, sigtimedwait, sigwait, sigwaitinfo, waitpid
timerLib	sleep, nanosleep

Caveat: due to the implementation of some of the I/O drivers in VxWorks, it is possible that a thread cancellation request can not actually be honored.

SUMMARY MATRIX

pthread **function**	**Implemented?**	**Note(s)**
pthread_atfork	Restricted	1
pthread_attr_destroy	Yes	
pthread_attr_getdetachstate	Yes	
pthread_attr_getguardsize	Yes	
pthread_attr_getinheritsched	Yes	
pthread_attr_getname	Yes	5
pthread_attr_getopt	Yes	5
pthread_attr_getschedparam	Yes	
pthread_attr_getschedpolicy	Yes	
pthread_attr_getscope	Yes	
pthread_attr_getstackaddr	Yes	
pthread_attr_getstacksize	Yes	
pthread_attr_getstack	Yes	

pthread **function**	**Implemented?**	**Note(s)**
pthread_attr_init	Yes	
pthread_attr_setdetachstate	Yes	
pthread_attr_setguardsize	Yes	
pthread_attr_setinheritsched	Yes	
pthread_attr_setname	Yes	5
pthread_attr_setopt	Yes	5
pthread_attr_setschedparam	Yes	
pthread_attr_setschedpolicy	Yes	
pthread_attr_setscope	Yes	2
pthread_attr_setstackaddr	Yes	
pthread_attr_setstacksize	Yes	
pthread_attr_setstack	Yes	
pthread_barrierattr_destroy	No	
pthread_barrierattr_getpshared	No	
pthread_barrierattr_init	No	
pthread_barrierattr_setpshared	No	
pthread_barrier_destroy	No	
pthread_barrier_init	No	
pthread_barrier_wait	No	
pthread_cancel	Yes	4
pthread_cleanup_pop	Yes	
pthread_cleanup_push	Yes	
pthread_condattr_destroy	Yes	
pthread_condattr_getclock	Yes	
pthread_condattr_getpshared	No	3
pthread_condattr_init	Yes	
pthread_condattr_setclock	Yes	
pthread_condattr_setpshared	No	3
pthread_cond_broadcast	Yes	
pthread_cond_destroy	Yes	
pthread_cond_init	Yes	
pthread_cond_signal	Yes	
pthread_cond_timedwait	Yes	
pthread_cond_wait	Yes	
pthread_create	Yes	
pthread_detach	Yes	
pthread_equal	Yes	
pthread_exit	Yes	
pthread_getconcurrency	Yes	
pthread_getcpuclockid	Yes	
pthread_getschedparam	Yes	
pthread_getspecific	Yes	
pthread_join	Yes	
pthread_key_create	Yes	

pthread **function**	**Implemented?**	**Note(s)**
pthread_key_delete	Yes	
pthread_kill	Yes	
pthread_mutexattr_destroy	Yes	
pthread_mutexattr_getprioceiling	Yes	
pthread_mutexattr_getprotocol	Yes	
pthread_mutexattr_getpshared	No	3
pthread_mutexattr_gettype	Yes	
pthread_mutexattr_init	Yes	
pthread_mutexattr_setprioceiling	Yes	
pthread_mutexattr_setprotocol	Yes	
pthread_mutexattr_setpshared	No	3
pthread_mutexattr_settype	Yes	
pthread_mutex_destroy	Yes	
pthread_mutex_getprioceiling	Yes	
pthread_mutex_init	Yes	
pthread_mutex_lock	Yes	
pthread_mutex_setprioceiling	Yes	
pthread_mutex_timedlock	Yes	
pthread_mutex_trylock	Yes	
pthread_mutex_unlock	Yes	
pthread_once	Yes	
pthread_rwlockattr_destroy	No	
pthread_rwlockattr_getpshared	No	
pthread_rwlock_destroy	No	
pthread_rwlock_init	No	
pthread_rwlock_rdlock	No	
pthread_rwlock_timedrdlock	No	
pthread_rwlock_timedwrlock	No	
pthread_rwlock_tryrdlock	No	
pthread_rwlock_trywrlock	No	
pthread_rwlock_unlock	No	
pthread_rwlock_wrlock	No	
pthread_self	Yes	
pthread_setcancelstate	Yes	
pthread_setcanceltype	Yes	
pthread_setconcurrency	Yes	
pthread_setschedparam	Yes	
pthread_setschedprio	Yes	
pthread_setspecific	Yes	
pthread_sigmask	Yes	
pthread_spin_destroy	No	
pthread_spin_init	No	
pthread_spin_lock	No	
pthread_spin_trylock	No	

pthread **function**	**Implemented?**	**Note(s)**
pthread_spin_unlock	No	
pthread_testcancel	Yes	

NOTES

1. The **pthread_atfork( )** function is implemented but always returns **ERROR** since **fork( )** is not available in VxWorks' user-side execution environment.
2. The contention scope thread scheduling attribute is always **PTHREAD_SCOPE_SYSTEM**, since threads (i.e. tasks) contend for resources with all other threads in the system.
3. The routines **pthread_condattr_getpshared( )**, **pthread_condattr_setpshared( )**, **pthread_mutexattr_getpshared( )** and **pthread_mutexattr_setpshared( )** are not currently supported by the VxWorks Real Time Process model.
4. Thread cancellation is supported in appropriate *pthread* routines and those routines already supported by VxWorks. However, the complete list of cancellation points specified by POSIX is not supported because routines such as **pselect( )** and **tcdrain( )** are not implemented by the user libraries of VxWorks.
5. VxWorks-specific routines provided as an extension to IEEE Std 1003.1 in order to handle VxWorks tasks' attributes.
6. VxWorks does not support multi-level scheduling; the **pthread_setconcurrency( )** and **pthread_getconcurrency( )** functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the level parameter is saved when **pthread_setconcurrency( )** is called so that a subsequent call to **pthread_getconcurrency( )** shall return the same value.

INCLUDE FILES

pthread.h

SEE ALSO

taskLib, **semMLib**, the VxWorks programmer guides.

pxTraceLib

NAME

pxTraceLib – POSIX trace user-level library

ROUTINES

posix_trace_attr_init( ) – initialize a POSIX trace attributes structure
posix_trace_attr_destroy( ) – destroy POSIX trace attributes structure
posix_trace_attr_getclockres( ) – copy clock resolution from trace attributes
posix_trace_attr_getcreatetime( ) – copy stream creation time to struct timespec
posix_trace_attr_getgenversion( ) – copy generation version from trace attributes
posix_trace_attr_getname( ) – copy stream name from trace attributes
posix_trace_attr_setname( ) – set the stream name in trace attributes
posix_trace_attr_getlogfullpolicy( ) – get log full policy from trace attributes

posix_trace_attr_setlogfullpolicy( ) – set log full policy in trace attributes
posix_trace_attr_getlogsize( ) – retrieve the size of the log for events
posix_trace_attr_setlogsize( ) – set the size of event data in a log
posix_trace_attr_getstreamfullpolicy( ) – get stream full policy
posix_trace_attr_setstreamfullpolicy( ) – set stream full policy
posix_trace_attr_getmaxdatasize( ) – get the maximum data size for an event
posix_trace_attr_setmaxdatasize( ) – set the maximum user event data size
posix_trace_attr_getmaxsystemeventsize( ) – get maximum size of a system event
posix_trace_attr_getmaxusereventsize( ) – get the maximum size of user event
posix_trace_attr_setstreamsize( ) – set size of memory to be used for event data
posix_trace_attr_getstreamsize( ) – get the size of memory used for event data
posix_trace_create( ) – create a trace stream without a log
posix_trace_create_withlog( ) – create a trace stream with a log file
posix_trace_shutdown( ) – stop tracing and destroy the stream
posix_trace_flush( ) – flush trace stream contents to trace log
posix_trace_clear( ) – reinitialize a trace stream
posix_trace_start( ) – start tracing using a pre-existing trace object
posix_trace_stop( ) – stop tracing
posix_trace_event( ) – record an event
posix_trace_eventid_open( ) – retrieve an event id for the supplied name
posix_trace_trid_eventid_open( ) – retrieve an event id for the supplied name
posix_trace_eventid_equal( ) – compare two event ids
posix_trace_eventtypelist_getnext_id( ) – retrieve an event id for a stream
posix_trace_eventtypelist_rewind( ) – reset the event id list iterator
posix_trace_eventid_get_name( ) – retrieve the name for a POSIX event id
posix_trace_getnext_event( ) – retrieve an event from a stream
posix_trace_timedgetnext_event( ) – retrieve an event from a stream, with timeout
posix_trace_trygetnext_event( ) – try to retrieve an event from a stream
posix_trace_get_filter( ) – get the event filter set from a stream
posix_trace_set_filter( ) – set the event filter associated with a stream
posix_trace_get_status( ) – retrieve the status of a stream
posix_trace_get_attr( ) – get the status of a trace stream
posix_trace_close( ) – close a pre-recorded trace stream
posix_trace_open( ) – create a stream from a pre-recorded trace log
posix_trace_rewind( ) – read the next event from the start of the trace
posix_trace_eventset_add( ) – add a POSIX trace event id to an event set
posix_trace_eventset_del( ) – remove a POSIX trace event id from an event set
posix_trace_eventset_ismember( ) – test whether a POSIX trace event is in a set
posix_trace_eventset_empty( ) – remove all events from an event set
posix_trace_eventset_fill( ) – fill an event set with a set of events

DESCRIPTION This library provides tracing functions according to the POSIX specification.

INCLUDE FILES **trace.h**

rngLib

NAME

rngLib – ring buffer subroutine library

ROUTINES

rngCreate( ) – create an empty ring buffer
rngDelete( ) – delete a ring buffer
rngFlush( ) – make a ring buffer empty
rngBufGet( ) – get characters from a ring buffer
rngBufPut( ) – put bytes into a ring buffer
rngIsEmpty( ) – test if a ring buffer is empty
rngIsFull( ) – test if a ring buffer is full (no more room)
rngFreeBytes( ) – determine the number of free bytes in a ring buffer
rngNBytes( ) – determine the number of bytes in a ring buffer
rngPutAhead( ) – put a byte ahead in a ring buffer without moving ring pointers
rngMoveAhead( ) – advance a ring pointer by *n* bytes

DESCRIPTION

This library provides routines for creating and using ring buffers, which are first-in-first-out circular buffers. The routines simply manipulate the ring buffer data structure; no kernel functions are invoked. In particular, ring buffers by themselves provide no task synchronization or mutual exclusion.

However, the ring buffer pointers are manipulated in such a way that a reader task (invoking **rngBufGet( )**) and a writer task (invoking **rngBufPut( )**) can access a ring simultaneously without requiring mutual exclusion. This is because readers only affect a *read* pointer and writers only affect a *write* pointer in a ring buffer data structure. However, access by multiple readers or writers *must* be interlocked through a mutual exclusion mechanism (i.e., a mutual-exclusion semaphore guarding a ring buffer).

This library also supplies two macros, **RNG_ELEM_PUT** and **RNG_ELEM_GET**, for putting and getting single bytes from a ring buffer. They are defined in **rngLib.h**.

```
int RNG_ELEM_GET (ringId, pch, fromP)
int RNG_ELEM_PUT (ringId, ch, toP)
```

Both macros require a temporary variable *fromP* or *toP*, which should be declared as **register int** for maximum efficiency. **RNG_ELEM_GET** returns 1 if there was a character available in the buffer; it returns 0 otherwise. **RNG_ELEM_PUT** returns 1 if there was room in the buffer; it returns 0 otherwise. These are somewhat faster than **rngBufPut( )** and **rngBufGet( )**, which can put and get multi-byte buffers.

INCLUDE FILES

rngLib.h

rtld

NAME

rtld – the dynamic linker

ROUTINES

dlclose( ) – unlink the shared object from the RTP's address space
dlerror( ) – get most recent error on a call to a dynamic linker routine
dlopen( ) – map the named shared object into the RTP's address space
dlsym( ) – resolve the symbol defined in the shared object to its address

DESCRIPTION

This library provides services to load shared libraries and plugins into the execution context of an RTP. The routines are not in a separate library but are included in every dynamically linked program automatically.

INCLUDE FILE

dlfcn.h

rtpLib

NAME

rtpLib – Real Time Process (RTP) facilities

ROUTINES

rtpSpawn( ) – spawns a new Real Time Process (RTP) in the system (syscall)
_exit( ) – terminate the calling process (RTP) (syscall)
rtpExit( ) – terminate the calling process
getpid( ) – Get the process identifier for the calling process (syscall)
getppid( ) – Get the parent process identifier for the calling process (syscall)
waitpid( ) – Wait for a child process to exit, and return child exit status
rtpInfoGet( ) – Get specific information on an RTP (syscall)
syscall( ) – invoke a system call using supplied arguments and system call number

DESCRIPTION

This library provides the interfaces to the Real Time Process (RTP) feature. Real Time Process is an optional feature of the VxWorks kernel that provides a process-like environment for applications. In the RTP environment, applications are protected and isolated from each other.

The Real Time Process feature offers the following types of protection:

- protection of the kernel from errant application code
- run-time isolation of applications from each other
- text and read-only data protection
- automatic resource reclamation
- **NULL** pointer access detection

An RTP is an active entity that always contains active tasks. An RTP may not exist without tasks.

ENABLING RTP SUPPORT

To enable RTP support, configure VxWorks with the **INCLUDE_RTP** component. This component includes all the functionalities contained in this library and all facilities necessary to support RTP.

To enable monitoring of RTPs, the component, **INCLUDE_RTP_SHOW**, must be configured in conjunction with **INCLUDE_RTP**.

CONFIGURATION

RTPs can be configured at creation time via **rtpSpawn( )**'s parameters as explained later in this manual and in **rtpSpawn( )**'s manual. It is also possible to change the default configuration parameters when the VxWorks image is generated (using Workbench's kernel configuration utility, or the vxprj command line utility). The new default values apply then to all RTPs. These configuration parameters, described in the component description file 01rtp.cdf, are:

RTP_KERNEL_STACK_SIZE
: Size of the kernel stack for user tasks.

KERNEL_HEAP_SIZE
: Size of the heap reserved to the kernel when RTPs are used in the system.

RTP_HOOK_TBL_SIZE
: Number of entries in the RTP create/delete hook tables.

SYSCALL_HOOK_TBL_SIZE
: Number of entries in the system call hook tables.

RTP_HEAP_INIT_SIZE
: Initial size of the RTP's heap. This can be overriden by the environment variable **HEAP_INITIAL_SIZE**.

RTP_SIGNAL_QUEUE_SIZE
: Maximum number of queued signal for a RTP. Note that POSIX requires that this number be at least 32.

RTP CREATION

Real Time Processes are created using the **rtpSpawn( )** API.

```
rtpSpawn (const char *rtpFileName, const char *argv[],
          const char *envp[], int priority, int uStackSize,
     int options, int taskOptions);
```

All RTPs are named and the names are associated with the *rtpFileName* argument passed to the **rtpSpawn( )** API.

All RTPs are created with an initial task which is also named after the *rtpFileName* argument passed to the **rtpSpawn( )** API: "i*Filename*", where *Filename* is made of the first 30 letters of the file name, excluding the extension.

The creation of an RTP will allocate the necessary memory to load the executable file for the application as well as for the stack of the initial task. Memory for the application is allocated from the global address space and is unique in the system. The memory of an RTP is not static; additional memory may be allocated from the system dynamically after the RTP has been created.

File descriptors are inherited from the caller, but the environment variables are not. If the application is expecting specific environment variables, an environment array must be created and passed to the **rtpSpawn( )** API. If the all of the caller's environment variables must be passed to the RTP, the *environ* variable can be used for this purpose (see example below).

The initial task starts its life as a task executing kernel code in supervisor mode. Once the application's code is loaded, the initial task switches to user mode and begins the execution of the application starting at the **_start( )** routine (ELF executable's entry point). The initial task initializes the user libraries and invokes all constructors in the application before executing the application's user code. The first user routine in the application is the **main( )** function and this function is called after all initializers and constructors are called. All C or C++ applications must provide a **main( )** routine. Its complete prototype is as follows:

```
int main
    (
    int       argc,   // number of arguments
    char * argv[], // NULL terminated array of arguments
    char * envp[], // NULL terminated array of environment strings
    void * auxp       // implementation specific auxiliary vector
    )
```

Note that, by convention, only the first two parameters are compulsory:

```
int main
    (
    int       argc,  // number of arguments
    char * argv[] // NULL terminated array of arguments
    }
```

There are attributes that may be set to customize the behavior of the RTP during **rtpSpawn( )** (including for example, whether symbol information is to be loaded, the initial task should be stopped at the entry point, or the priority and task options of the initial task.) The reference entry for **rtpSpawn( )** provides more details on the options and other configuration parameters available when creating an RTP.

RTP TERMINATION

Real Time Process are terminated in several ways:

- Calling **exit( )** within the RTP. This includes the initial task of the RTP reaching the end of its execution.
- When the last task of the RTP exits.
- A fatal **kill( )** signal is sent to an RTP.
- An unrecoverable exception occurs.

The termination of an RTP will delete the RTP executable and return all memory (virtual and physical memory) used by it to the system. System objects allocated and owned by the RTP will also be deleted from the system. (See **objLib** reference entry for more details on object resource reclamation.) Memory mapped to the RTP will also be freed back into the system. Note that public objects still in use by other users in the system will be inherited by the kernel, and will not be reclaimed at this point.

Any routines registered with the **atexit( )** function will be called in the reverse order that they are registered. These **atexit( )** routines will be called in a normal termination of an RTP. Abnormal termination of an RTP, such as invoking the deletion from the kernel or sending a fatal **kill( )** signal to an RTP, will not cause the **atexit( )** routines to be called.

RTP INITIALIZATION

Real Time Processes (RTPs) may be initialized in various ways: automatically by the system during boot time using the RTP startup facility, by launching them from the shell(s), or programmatically using the **rtpSpawn( )** API. The automatic initialization is available in two forms:

- Using the **INCLUDE_RTP_APPL_USER** component that enables users to write their own code to spawn their RTPs and to pass parameters to the RTP.
- Using the startup script (s field) in the boot parameters. Users may overload the startup script field to specify RTPs and their parameters to be called at system boot time. The format to use is the following:

  ```
  startup script (s): #print.vxe^"%s\n"^"hello"#
  ```

  One or more RTPs may be set up in the startup script field. The **#** character is the delimiter for each RTP and the **^** is the delimiter for the parameters of the RTP.

RTPs may be spawned and initialized from the shell(s):

- Using the traditional C interpreter: the **rtpSp( )** command will allow the user to execute a VxWorks executable file and pass arguments to its **main( )** routine.

  ```
  -> rtpSp "myVxApp.vxe first second third"
  ```

- Using the RTP command shell by either directly typing the path and name of the executable file and then the list of arguments (similar to a UNIX shell) or use the rtp exec command. **help rtp** on the command shell will provide more details.

  ```
  [vxWorks *]# /home/myVxApp.vxe first second third

  OR

  [vxWorks *]# rtp exec /home/myVxApp.vxe first second third
  ```

- Programmatically, from a kernel task or an other RTP, using the **rtpSpawn( )** API:

```
    const  char * args[] = {"/romfs/myApp.vxe", "-arg1", "-arg2 0x1000",
NULL};
    ...
    rtpSpawn (args[0], args, NULL, 100, 0x10000, 0, VX_FP_TASK);
```

or (when the caller's environment variables must be passed to the application):

```
rtpSpawn (args[0], args, environ, 100, 0x10000, 0, VX_FP_TASK);
```

Note that the *environ* variable is available in the RTP space only. In the kernel the **envGet( )** API is to be used instead. Note also that a specific set of environment variables can be programmatically passed to a RTP via its *envp* parameter:

```
const char * envp[] = {"MY_ENV_VAR1=foo", "MY_ENV_VAR2=bar", NULL};
...
rtpSpawn (args[0], args, envp, 100, 0x10000, 0, VX_FP_TASK);
```

TASKS

Every task in the system will have an owner, whether it is the kernel or an RTP. This owner is also the owner of the task object (tasks are <WIND objects>). Unlike other objects, the ownership of a task is restricted to the task's RTP or the kernel. This restriction exists since the task's stack will be allocated from the RTP's memory resources.

By default, tasks running outside the kernel run in the CPU's *user* mode. A task will run in the CPU's *supervisor* mode (**VX_SUPERVISOR_MODE** option is set for the task), if the task is created in the kernel.

The scheduling of tasks is not connected in any way with the RTP that owns them. Even when RTPs are configured into the operating system, tasks are still scheduled based on their priorities and readiness to execute. Note that in the specific case when POSIX threads are executed in the RTP it is mandatory that the POSIX scheduler be used in the system (**INCLUDE_POSIX_PTHREAD_SCHEDULER** component).

Unlike kernel tasks, user tasks (i.e. tasks created in the RTP) cannot have their own private environment variables. They all share the RTP's environment.

Note also that the initial task of a RTP cannot be restarted (see **taskRestart( )** for details).

SHARING DATA

The real time process model also supports the sharing of data between RTPs. This sharing can be done using shared data regions. Refer to the **sdLib** reference entries for more information on shared data regions.

To simply share memory, or memory-mapped I/O, with another RTP, a shared data region needs to be created. Then, the *client* RTP (i.e. the one wishing to access the shared resource) simply needs to map the shared data region into its memory space. This is achieved using the **sdMap( )** function. See the reference entry for the **sdMap( )** function for more information about creating shared data mappings. This sharing relationship must be created at run-time by the application.

SHARING CODE

Sharing of code between RTPs are done using shared libraries. Shared libraries are dynamically loaded at runtime by the RTPs that reference them.

To use shared libraries, the RTP executable must specify at build time that it wants to resolves its undefined symbols using shared libraries. The location of the shared libraries must be provided to the RTP executable using one of the following:

- the -rpath *path* compiler flag

- setting the environment variable **LD_LIBRARY_PATH** for the RTP

If the above two options are not used, the location of the RTP executable will be used to find the shared libraries.

For more information on how to use shared libraries, see the VxWorks programmer guides.

RTP STATES

An RTP life cycle revolves around the following states:

RTP_STATE_CREATE
When an RTP object is created it's initial state is **RTP_STATE_CREATE**. It remains in the state until the RTP object is fully initialized, the image loaded into RTP memory space and the initial task is about to transition to user mode. If initialization is successful, the state transitions to **RTP_STATE_NORMAL** otherwise it transitions to **RTP_STATE_DELETE**.

RTP_STATE_NORMAL
This is the state that indicates that the RTP image is fully loaded and tasks are running in user mode. When the RTP terminates it transitions to **RTP_STATE_DELETE**.

RTP_STATE_DELETE
This is the state that indicates that the RTP is being deleted. No further operations can be performed on the RTP in this state. Once the deletion is complete, the RTP object and it's resources are reclaimed by the kernel.

All RTP operations can be done only when the RTP is in **RTP_STATE_CREATE** or **RTP_STATE_NORMAL** state.

RTP STATUS

RTP status bits indicates some important events happening in the RTP life cycle:

RTP_STATUS_STOP
This status bit is set when a stop signal is sent to the RTP. All tasks within the RTP are stopped. A SIGCONT signal sent to the stopped RTP resumes all stopped tasks within the RTP, thus unsetting this bit.

RTP_STATUS_ELECTED_DELETER
This status bit is set once a task is selected to delete the RTP among competing deleting tasks. The RTP is now destined to die. The RTP delete hooks are called after this election, but before the RTP state goes to **RTP_STATE_DELETE**. Once the RTP transitions to **RTP_STATE_DELETE**, this bit is unset.

SYSTEM CALL BUFFER VALIDATION

By default any user buffer passed to a system call will be validated to ensure that it belongs to the RTP's memory space. This validation is a lengthy operation which adds to the system call overhead. The buffer validation can be turned off for a specific RTP by spawning it with the option **RTP_BUFFER_VAL_OFF** (0x20) set. However this leaves a potential security hole so this option should be used only once the application code is properly debugged.

SMP CONSIDERATIONS

By default RTP tasks inherit the CPU affinity setting of the task that created the RTP. If the parent task has no specific CPU affinity (i.e. it can execute on any available CPU and may migrate from one CPU to the other during its lifetime) then the RTP's tasks have no specific CPU affinity either. If the parent task has its affinity set to a given CPU then, by default, the RTP tasks inherit this affinity and execute only on the same CPU as the RTP's parent task.

By using the **rtpSpawn( )**'s option **RTP_CPU_AFFINITY_NONE** it is possible to create a RTP which tasks have no specific CPU affinity even though the RTP's parent task may have a specific CPU affinity.

INCLUDE FILES **rtpLib.h**

SEE ALSO **sigLib**, **edrLib**, **sdLib**

salClient

NAME **salClient** – socket application client library

ROUTINES

salOpen( ) – establish communication with a named socket-based server
salSocketFind( ) – find sockets for a named socket-based server
salNameFind( ) – find services with the specified name
salCall( ) – invoke a socket-based server

DESCRIPTION

This portion of the Socket Application Library (SAL) provides the infrastructure for implementing a socket-based client application. The routines provided by SAL allow client applications to communicate easily with socket-based server applications that are registered with the Socket Name Service (SNS). Some routines can also be used to communicate with unregistered server applications. SAL routines assume connection oriented message based communications. Although it could provide support for all protocols with the above features, the current implementation is supporting only local (single node) inter process communication using the COMP (Connection Oriented Message Passing) protocol and distributed (multi-node) inter process communication using the TIPC (Transparent Inter-Process Communication) protocol.

SAL Client

The SAL client API allows a client application to communicate with a specified server application by using socket descriptors. A client application can utilize SAL routines to communicate with different server applications in succession, or create multiple SAL clients that are each linked to a different server.

A client application typically calls **salOpen( )** to configure a socket descriptor associated with a named server application. **salOpen( )** simplifies the procedures needed to initialize the socket and its connection to the server. The server can be easily identified by a name, represented by a character string. The client application can then communicate with the server by passing the socket descriptor to standard socket API routines, such as **send( )** and **recv( )**. Alternatively, the client application can perform a **send( )** and **recv( )** as a single operation using **salCall( )**. When the client application no longer needs to communicate with a server it calls **close( )** to close the socket to the server.

A client application can utilize **salSocketFind( )** to exercise more control over the establishment of communication with a server, as an alternative to using **salOpen( )**. **salSocketFind( )** can be used to determine the socket addresses related to a server, and then create a socket to communicate with the server. The client can therefore choose the server socket address or addresses that better suits its needs. A client can also use **salNameFind( )** to identify one or more services based on a search pattern. Therefore, the client does not need to know the exact name of a service and, in case multiple names are found, it can choose which ones to use.

Because normal socket descriptors are used, the client application also has access to all of the standard socket API.

EXAMPLE

The following code illustrates how to create a client that utilizes an "ping" service which simply returns each incoming message to the sender. The maximum size of a message is limited to **MAX_PING_SIZE** bytes. This service uses the connection-based COMP socket protocol.

```
/* This routine creates and runs a client of the ping service. */

#include "vxWorks.h"
#include "dsi/salClient.h"

#define MAX_PING_SIZE 72

STATUS pingClient
    (
    char * message,                         /* message buffer */
    int    msgSize                          /* size of message */
    )
    {
    char reply[MAX_PING_SIZE];              /* reply buffer */
    int replySize;                          /* size of reply */
    int sockfd;                             /* socket file descriptor */

    /* set up client connection to PING server */

    if ((sockfd = salOpen ("ping")) < 0)
        {
        return ERROR;
        }

    /* send message to PING server and get reply */

    replySize = salCall (sockfd, message, msgSize,
                         reply, sizeof (reply));
```

```
        /* tear down client connection to PING server */

        if (close (sockfd) <0)
            return ERROR;

        /* check that reply matches message */

        if ((replySize != msgSize) || (memcmp (message, reply, msgSize) !=
0))
            {
            return ERROR;
            }

        return OK;
        }
```

CONFIGURATION To use the SAL client library, configure VxWorks with the **INCLUDE_SAL_CLIENT** component.

INCLUDE FILES **salClient.h**

SEE ALSO **salServer**, **snsLib**

salServer

NAME **salServer** – socket application server library

ROUTINES
salCreate( ) – create a named socket-based server
salDelete( ) – delete a named socket-based server
salServerRtnSet( ) – configures the processing routine with the SAL server
salRun( ) – activate a socket-based server
salRemove( ) – Remove service from SNS by name

DESCRIPTION This portion of the Socket Application Library (SAL) provides the infrastructure for implementing a socket-based server application. The data structures and routines provided by SAL allow the application to communicate easily with socket-based client applications that locate the server using the Socket Name Service (SNS).

SAL Server ID

The "SAL Server ID" refers to an internal data structure that is used by many routines in the SAL server library. The server data structure allows a server application to provide service to any number of client applications. A server application normally utilizes a single SAL server in its main task, but it is free to spawn additional tasks to handle the processing for individual clients if parallel processing of client requests is required.

Main Capabilities

A server application typically calls **salCreate( )** to configure a SAL server with one or more sockets that are then registered with SNS under a specified service identifier. The number of sockets created depends on which address families, socket types, and socket protocols are specified by the server application. The current implementation supports only connection-oriented message based socket types. Although it could provide support for all protocols with the above features, the current implementation is supporting both local (single node) inter process communication using the COMP (Connection Oriented Message passing) protocol and distributed (multi-node) inter process communication using the TIPC (Transparent Inter-Process Communication) protocol. The socket addresses used for the server's sockets are selected automatically and cannot be specified by the server application using **salCreate( )**.

Once created, a SAL server must be configured with one or more processing routines before it is activated.

- The "accept" routine is invoked whenever an active socket is created as the result of a new client connecting to the server.
- The "read" routine is invoked whenever an active socket is ready for reading or can no longer be read.

Configuring of the processing routines is accomplished by calling the **salServerRtnSet( )** function.

If no routine is supplied, the service will not be activated.

Activation of a SAL server is accomplished by calling **salRun( )**. A SAL server runs indefinitely once it has been activated, monitoring the activities on its connections and calling the appropriate processing routines as needed. The SAL server becomes deactivated only at the request of the server application (through the processing routines) or if an unexpected error is detected by **salRun( )**.

Once a SAL server has been deactivated the server application calls **salDelete( )** to close the server's sockets and deregister the service identifier from SNS.

Processing Routines

The "accept" routine is utilized by any server application that incorporates passive (i.e. listening) sockets into the SAL server. The routine should determine if the connection should be accepted and the new socket added to the SAL server. The routine can return the following values:

SAL_SOCK_KEEP
the SAL server has accepted the new connection and the new socket should be added to the SAL server.

SAL_SOCK_CLOSE
the routine is requesting the SAL server to close the socket.

SAL_SOCK_IGNORE
the SAL server will not add the new socket but it will not close it. This could be because the user application is going to have the socket managed by another task or because it has already closed the socket.

Any other value is considered as an error and deactivates the SAL server.

If a SAL server is not configured with an accept routine **salRun( )** uses a default routine that automatically approves of the socket and adds it to the server.

The "read" routine is utilized by any server application that incorporates active sockets into the SAL server. The routine should read the specified socket and process the input accordingly, possibly generating a response. The read routine should return an appropriate value to let **salRun( )** know what to do with the socket or to the SAL server.

SAL_SOCK_CLOSE
the SAL server closes the socket and removes it the from server.

SAL_SOCK_IGNORE
the SAL server removes the socket from the list without closing it. This might be useful when the application requires another task to take care of the socket.

SAL_SOCK_KEEP
the socket is kept in the SAL server.

SAL_RUN_TERMINATE
salRun( ) is terminated, with an **OK** return value. The sockets are not closed.

Any other value is considered as an error and deactivates the SAL server.
The read routine should close the socket and return **SAL_SOCK_IGNORE**, or ask the SAL server to close the socket (by returning **SAL_SOCK_CLOSE**), if it detects that the socket connection has been closed by the client. This state is normally indicated by a read operation that receives zero bytes.

If a SAL server is not configured with a read routine and active sockets are present, **salRun( )** uses a default routine that deactivates the server with an error.

NOTE

Care must be taken to ensure that a processing routine does not cause **salRun( )** to block, otherwise the actions of a single client can halt the server's main task and thereby deny use of the server to other clients. One solution is to use the **MSG_DONTWAIT** flag when reading or writing an active socket; an alternative solution is to use a distinct task for each active socket and not incorporate them into the SAL server.

EXAMPLE

The following code illustrates how to create a server that implements an "ping" service which simply returns each incoming message to the sender. The service satisfies the first **MAX_REQ_COUNT** requests only. Once it has reached the threshold it terminates. The maximum size of a message is limited to **MAX_PING_SIZE** bytes. This service uses the connection-based COMP socket protocol.

```
#include "vxWorks.h"
#include "sockLib.h"
```

```
#include "dsi/salServer.h"

/* defines */

#define MAX_PING_SIZE 72      /* max message size */
#define MAX_REQ_COUNT 5       /* max number of client requests */

/* forward declarations */

LOCAL SAL_RTN_STATUS  pingServerRead (int sockfd, void * pData);

/* This routine creates and runs the server for the ping service. */

STATUS pingServer (void)
    {
    SAL_SERVER_ID serverId;                  /* server structure */
    STATUS result;                           /* return value */
    int count;                               /* counter */

    /* create server socket & register service with SNS */

    if ((serverId = salCreate ("ping",AF_LOCAL, SOCK_SEQPACKET, 0,
                   NULL, 0))  == NULL)
        {
        return ERROR;
        }

    /* configure read routine for server */

    salServerRtnSet (serverId, SAL_RTN_READ, pingServerRead);

    /* request counter initialized */

    count = 0;

    /* activate the server (never returns unless a fatal error occurs */
    /* or the application processing routine requests a termination) */

    result = salRun (serverId, &count);

    /* close server socket & deregister service from SNS */

    salDelete (serverId);

    return result;
    }


/* This is the read routine for the ping server. */

LOCAL SAL_RTN_STATUS  pingServerRead
    (
    int sockfd,                              /* active socket to read */
    void * pData                             /* user data */
    )
    {
```

```
char message[MAX_PING_SIZE];            /* buffer for message */
int msgSize;                            /* size of message */
int * pCounter;                         /* request counter */

/* get message from specified client */

msgSize = recv (sockfd, message, sizeof (message), MSG_DONTWAIT);

if (msgSize <= 0)
    {
    /* client connection has been closed by client or has failed */

    return SAL_SOCK_CLOSE;
    }

/* send message back to client */

if (send (sockfd, message, msgSize, MSG_DONTWAIT) < 0)
    {
    /* client connection has failed */

    close (sockfd);
    return SAL_SOCK_IGNORE;
    }

pCounter = pData;

if (*pCounter++ >= MAX_REQ_COUNT)
    return SAL_RUN_TERMINATE;

/* indicate that client connection is still OK */

return SAL_SOCK_KEEP;
}
```

CONFIGURATION To use the SAL server library, configure VxWorks with the **INCLUDE_SAL_SERVER** component.

INCLUDE FILES **salServer.h**

SEE ALSO **salClient**, **snsLib**

schedPxLib

NAME **schedPxLib** – scheduling library (POSIX)

ROUTINES
sched_setparam( ) – set a task's priority (POSIX)
sched_getparam( ) – get the scheduling parameters for a specified task (POSIX)

sched_setscheduler( ) – set scheduling policy and scheduling parameters (POSIX)
sched_getscheduler( ) – get the current scheduling policy (POSIX)
sched_yield( ) – relinquish the CPU (POSIX)
sched_get_priority_max( ) – get the maximum priority (POSIX)
sched_get_priority_min( ) – get the minimum priority (POSIX)
sched_rr_get_interval( ) – get the current time slice (POSIX)

DESCRIPTION

This library provides POSIX-compliance scheduling routines for VxWorks applications (Real Time Processes). The routines in this library allow the user to get and set priorities and scheduling schemes, get maximum and minimum priority values, and get the time slice if round-robin scheduling is enabled.

When making task priority changes from a task running in user mode, the changes can only be made on tasks running within the context of the current Real Time Process; i.e. it is not possible to change the priority of tasks belonging to another RTP, or tasks within the kernel.

The POSIX standard specifies a priority numbering scheme in which higher priorities are indicated by larger numbers. The VxWorks native numbering scheme is the reverse of this, with higher priorities indicated by smaller numbers. For example, in the VxWorks native priority numbering scheme, the highest priority task has a priority of 0.

In VxWorks, POSIX scheduling interfaces are implemented using the POSIX priority numbering scheme. This means that the priority numbers used by this library *do not* match those reported and used in all the other VxWorks components. It is possible to change the priority numbering scheme used by this library by setting the global variable **posixPriorityNumbering**. If this variable is set to **FALSE**, the VxWorks native numbering scheme (small number = high priority) is used, and priority numbers used by this library will match those used by the other portions of VxWorks.

The routines in this library are compliant with POSIX 1003.1b. In particular, application task priorities are set and reported through the structure **sched_setparam**, which has a single member:

```
struct sched_param              /* Scheduling parameter structure */
    {
    int sched_priority;         /* scheduling priority */
    };
```

POSIX 1003.1b specifies this indirection to permit future extensions through the same calling interface. For example, because **sched_setparam( )** takes this structure as an argument (rather than using the priority value directly) its type signature need not change if future schedulers require other parameters.

CONFIGURATION

This library requires the **INCLUDE_POSIX_SCHED** component to be configured into the kernel; *errno* may be set to **ENOSYS** if this component is not present.

INCLUDE FILES

sched.h

SEE ALSO

POSIX 1003.1b document, user-side **taskLib**

sdLib

NAME

sdLib – shared data facilities

ROUTINES

sdCreate( ) – Create a new shared data region
sdOpen( ) – Open a shared data region for use
sdDelete( ) – Delete a shared data region
sdMap( ) – Map a shared data region into an application or the kernel
sdUnmap( ) – Unmap a shared data region from an application or the kernel
sdProtect( ) – Change the protection attributes of a mapped shared data region
sdInfoGet( ) – Get specific information about a shared data region
_sdCreate( ) – Create a new shared data region (system call)
_sdOpen( ) – Open a shared data region for use (system call)

DESCRIPTION

This library contains details of functions related to Shared Data regions in VxWorks. The purpose of shared data regions is to allow physical memory, or other physical resources such as blocks of memory mapped I/O space to be shared between multiple applications.

CREATION

A shared data region can be created via one of two routines:

```
sdOpen (char * name, int options, int mode, UINT32 size,
     off_t physAddress, MMU_ATTR attr, void ** pVirtAddress);

sdCreate (char * name, int options, UINT32 size, off_t physAddress,
          MMU_ATTR attr, void ** pVirtAddress);
```

The behavior of sdOpen is determined by the value of its *mode* parameter. If the default value of 0 is passed, then a shared data region will not be created.

To create a shared data region using **sdOpen( )** the **OM_CREATE** flag must be passed in the *mode* parameter. If just this flag is passed in *mode* and a shared data region with the *name* specified does not already exist in the system the region will be created. However, if a shared data region *name* already exists, then **sdOpen( )** will map that region into the memory context of the calling task and return its **SD_ID**.

If both the **OM_CREATE** and **OM_EXCL** flags are passed in the *mode* parameter of **sdOpen( )**, then a new region will be created if a region with the *name* specified does not already exist in the system. If such a region does exist then no region will be created and **NULL** will be returned.

The behavior of **sdCreate( )** is identical to that of **sdOpen( )** with both the **OM_CREATE** and **OM_EXCL** flag specified in the *mode* parameter.

While it is possible to specify a physical location of a shared data region with the arguments *physAddress* and *size*, that address range must not be mapped into any other context in the system. No other restrictions are placed. If *physAddress* is **NULL** the system will allocate the physical memory from the available RAM. If there is not enough RAM available in the system the creation will fail and **NULL** will be returned.

It is not possible to specify a virtual location for a shared data region. The location of the region will be returned at *pVirtAddress*.

A *size* of greater than 0 must be specified to create a shared data region.

On creation the shared data region will be mapped into the memory context associated with the task which invoked the call. The shared data region will be owned by the RTP of that task. If the RTP that owns a shared data region exits the kernel will assume ownership of the region.

A shared data region is initially mapped into its owner's context with both read and write access privileges in addition to those specified by *attr*. This may be changed by either a call to **sdProtect( )** or **sdMap( )**. The MMU attribute value specified in *attr* will be the default value for the shared data region. This will also serve as the limit of access privileges all subsequent clients of the region may use. That is, if *attr* does not specify a particular attribute applications other than the owner will not have, nor be able to set, that attribute on the region within their memory context. For example, if *attr* is set to (**SD_ATTR_RW** | **SD_CACHE_OFF**) an application other than the owner may use **sdProtect( )** to restrict its access to (**SD_ATTR_RO** | **SD_CACHE_OFF**), but not to set its access to (**SD_ATTR_RWX** | **SD_CACHE_OFF**).

USING SHARED DATA

To access a shared data region from an application it must be initially be mapped to that application via a call to either **sdOpen( )** or **sdCreate( )**.

These routines return a **SD_ID** which may be used by any task within that application. A **SD_ID** may not be shared between applications or between an application and the kernel.

Once this initial mapping is done tasks in the application may access the memory as if it were local unless explicitly unmapped by a task in the application with a call to **sdUnmap( )**.

Task may call the following routines using the application's unique **SD_ID**:

sdDelete( )

sdMap( )

sdUnmap( )

sdProtect( )

sdInfoGet( )

By default each client application, excepting the owner, will have the access privileges specified by the value of *attr* at creation. However, an application may change its access privileges via a call to either **sdProtect( )** or **sdMap( )**, but will be limited to the default attributes of the region or a subset thereof. The owner of a region will by default have both read and write privileges in addition to the region default attributes and may change its local access rights to any valid combination. See **vmLib** for details on what valid values of *attr* are available.

It is important to note that the shared data region object provides no mutual exclusion. If more than one application, or the kernel and one application or more, require access to this region some form of mutual exclusion must be used.

A shared data region may be created that is private to the creator by passing the **SD_PRIVATE** option in the *options* field. No other application, including the kernel, will be able to map such a region.

DELETING SHARED DATA

When all applications have unmapped a shared data region, it may be deleted using the **sdDelete( )** function. This will return all resources associated with the region and remove it from the system. It is not possible to delete a shared data region that is still in use by an application or the kernel. To unmap a shared data region from an application it is necessary for a task in that application to call **sdUnmap( )**.

By default the last application to unmap a shared data region will force a deletion of the region. However, if the shared data region was created with the option **SD_LINGER** specified it will remain until explicitly deleted by calling **sdDelete( )**.

CONFIGURATION

To configure shared data management into the system, the component **INCLUDE_SHARED_DATA** must be included in the kernel; *errno* will be set to **ENOSYS** if this component is not present.

INCLUDE FILES

sdLib.h

SEE ALSO

rtpLib, **slLib**, **vmLib**, the VxWorks programmer guides.

semEvLib

NAME

semEvLib – VxWorks user events support for semaphores

ROUTINES

semEvStart( ) – start event notification process for a semaphore
semEvStop( ) – stop event notification process for a semaphore

DESCRIPTION

This library is an extension to **eventLib**, the events library. Its purpose is to support events for semaphores.

The functions in this library are used to control registration of tasks on a semaphore. The routine **semEvStart( )** registers a task and starts the notification process. The function **semEvStop( )** un-registers the task, which stops the notification mechanism.

When a task is registered and the semaphore becomes available, not taken immediately after being given, the events specified are sent to that task. However, if events are sent, there is no guarantee that the semaphore will still be available afterwards.

INCLUDE FILES **semEvLib.h**

SEE ALSO **eventLib**, **semLib**, the VxWorks programmer guides.

semInfo

NAME **semInfo** – user-level semaphore info routines

ROUTINES **semInfoGet( )** – get information about a semaphore

DESCRIPTION This library provides the routine **semInfoGet( )** to extract semaphore information.

INCLUDE FILES **semLib.h**

SEE ALSO the VxWorks programmer guides.

semLib

NAME **semLib** – user-level semaphore library

ROUTINES

semBCreate( ) – create and initialize a binary semaphore
semCCreate( ) – create and initialize a counting semaphore
semMCreate( ) – create and initialize a mutual-exclusion semaphore
semOpen( ) – open a named semaphore
semClose( ) – close a named semaphore
semUnlink( ) – unlink a kernel named semaphore
semDelete( ) – delete a semaphore
semFlush( ) – unblock every task pended on a semaphore
semGive( ) – give a semaphore
semTake( ) – take a semaphore
semExchange( ) – atomically give and take a pair of semaphores
_semOpen( ) – open a kernel semaphore (system call)
_semTake( ) – take a kernel semaphore (system call)
_semGive( ) – give a kernel semaphore (system call)
semCtl( ) – perform a control operation against a kernel semaphore (system call)

DESCRIPTION Semaphores are the basis for synchronization and mutual exclusion in VxWorks. They are powerful in their simplicity and form the foundation for numerous VxWorks facilities.

Different semaphore types serve different needs, and while the behavior of the types differs, their basic interface is the same. This library provides semaphore routines common to all VxWorks semaphore types. For all types, the two basic operations are **semTake( )** and **semGive( )**, the acquisition or relinquishing of a semaphore.

Mutex semaphores offer the greatest speed while binary semaphores offer the broadest applicability.

The **semLib** library provides all the semaphore operations, including routines for semaphore control, deletion, and information. Semaphores must be validated before any semaphore operation can be undertaken. An invalid semaphore ID results in **ERROR**, and an appropriate **errno** is set.

SEMAPHORE CONTROL

The **semTake( )** call acquires a specified semaphore, blocking the calling task or making the semaphore unavailable. All semaphore types support a timeout on the **semTake( )** operation. The timeout is specified as the number of ticks to remain blocked on the semaphore. Timeouts of **WAIT_FOREVER** and **NO_WAIT** codify common timeouts. If a **semTake( )** times out, it returns **ERROR**. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The **semGive( )** call relinquishes a specified semaphore, unblocking a pended task or making the semaphore available. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The **semFlush( )** call may be used to atomically unblock all tasks pended on a semaphore queue, i.e., all tasks will be unblocked before any are allowed to run. It may be thought of as a broadcast operation in synchronization applications. The state of the semaphore is unchanged by the use of **semFlush( )**; it is not analogous to **semGive( )**.

SEMAPHORE DELETION

The **semDelete( )** call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return **ERROR**. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that already has taken (owns) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully taken.

VXWORKS EVENTS

If a task has registered for receiving events with a semaphore, events will be sent when that semaphore becomes available. By becoming available, it is implied that there is a change of state. For a binary semaphore, there is only a change of state when a **semGive( )** is done on a semaphore that was taken. For a counting semaphore, there is always a change of state when the semaphore is available, since the count is incremented each time. For a mutex, a **semGive( )** can only be performed if the current task is the owner, implying that the semaphore has been taken; thus, there is always a change of state.

INCLUDE FILES

semLib.h

SEE ALSO **taskLib**, **semEvLib**, **eventLib**, the VxWorks programmer guides.

semPxLib

NAME **semPxLib** – user-level semaphore synchronization library (POSIX)

ROUTINES
sem_init( ) – initialize an unnamed semaphore (POSIX)
sem_destroy( ) – destroy an unnamed semaphore (POSIX)
sem_open( ) – initialize/open a named semaphore (POSIX)
sem_close( ) – close a named semaphore (POSIX)
sem_unlink( ) – remove a named semaphore (POSIX)
sem_wait( ) – lock (take) a semaphore, blocking if not available (POSIX)
sem_trywait( ) – lock (take) a semaphore, returning error if unavailable (POSIX)
sem_post( ) – unlock (give) a semaphore (POSIX)
sem_getvalue( ) – get the value of a semaphore (POSIX)
sem_timedwait( ) – lock (take) a semaphore with a timeout (POSIX)

DESCRIPTION This library implements the semaphore interface based on the POSIX 1003.1b specifications. For alternative semaphore routines designed expressly for VxWorks, see the reference entries for **semLib** and other semaphore libraries mentioned there. POSIX semaphores are counting semaphores; as such they are most similar to the **semCLib** VxWorks-kernel semaphores.

The main advantage of POSIX semaphores is portability (to the extent that alternative operating systems also provide these POSIX interfaces). However, VxWorks-specific semaphores provide the following features absent from the semaphores implemented in this library: priority inheritance, task-deletion safety, the ability for a single task to take a semaphore multiple times, ownership of mutual-exclusion semaphores, semaphore timeout, and the choice of queuing mechanism.

POSIX defines both named and unnamed semaphores; **semPxLib** includes separate routines for creating and deleting each kind. For other operations, applications use the same routines for both kinds of semaphore.

TERMINOLOGY The POSIX standard uses the terms *wait* or *lock* where *take* is normally used in VxWorks, and the terms *post* or *unlock* where *give* is normally used in VxWorks. VxWorks documentation that is specific to the POSIX interfaces (such as the remainder of this reference entry, and the reference entries for subroutines in this library) uses the POSIX terminology, in order to make it easier to read in conjunction with other references on POSIX.

SEMAPHORE DELETION

The **sem_destroy( )** call terminates an unnamed semaphore and deallocates any associated memory; the combination of **sem_close( )** and **sem_unlink( )** has the same effect for named semaphores. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that has already locked that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully locked. (Similarly, for named semaphores, applications should take care to close only semaphores that the closing task has opened.)

If there are tasks blocked waiting for the semaphore, **sem_destroy( )** fails and sets **errno** to **EBUSY**.

Detection of deadlock is not considered in this implementation.

SEMAPHORE NAME LENGTH

The semaphore namespace in VxWorks is not associated with the filesystem. Thus, it is incorrect to use the POSIX variable values **NAME_MAX** and **PATH_MAX** to specify the maximum length of the semaphore name and path. Instead, **_VX_PX_SEM_NAME_MAX** and **_VX_PX_SEM_PATH_MAX** are defined for the corresponding length limits for semaphore names. The semantic of the **_VX_PX_SEM_PATH_MAX** is the same as for **PATH_MAX**, and the semantic of the **_VX_PX_SEM_NAME_MAX** is the same as for **NAME_MAX**.

For the same reason, POSIX **pathconf( )** API must not be used with semaphore object path.

CONFIGURATION

This library requires the **INCLUDE_POSIX_SEM** component to be configured into the kernel; *errno* may be set to **ENOSYS** if this component is not present.

INCLUDE FILES

semaphore.h

SEE ALSO

POSIX 1003.1b document, **semLib**

semRWLib

NAME

semRWLib – user-level read/write semaphore library

ROUTINES

semRWCreate( ) – create and initialize a reader/writer semaphore
semWTake( ) – take a semaphore in write mode
semRTake( ) – take a semaphore as a reader

DESCRIPTION

This library provides the interface to VxWorks reader/writer semaphores. Reader/writer semaphores provide a method of synchronizing groups of tasks that can be granted concurrent access to a resource with those tasks that require mutually exclusive access to that resource. Typically this correlates to those tasks that intend to modify a resource and those which intend only to view it.

Like a mutual-exclusion semaphore the following restrictions exist:

- It can only be given by the task that took it.
- It may not be taken or given from interrupt level.
- The **semFlush( )** operation is illegal.

A reader/writer semaphore differs from other semaphore types in that a mode is specified by the choice of the "take" routine. It is this mode that determines whether the caller requires mutually exclusive access or if concurrent access would suffice.

The two modes are "read" and "write", and specified by calling one of the following routines:

semRTake( ) - take a semaphore in "read" mode

semWTake( ) - take a semaphore in "write" mode

For tasks that take a reader/writer semaphore in "write" mode the behavior is quite similar to a mutex semaphore. That task will own the semaphore exclusively.

If a timeout other than **NO_WAIT** is specified an attempt to acquire a reader/writer semaphore in "write" mode when the semaphore is held by another writer or any number of readers will result in the caller pending.

The behavior of a reader/writer semaphore when taken in "read" mode is unique. This does not imply exclusive access to a resource. In fact, a semaphore may be concurrently held in this mode by a number of tasks. These tasks can be seen as collectively owning the semaphore.

Mutual exclusion between a collection of reader tasks and all writer tasks will be maintained.

If a timeout other than **NO_WAIT** is specified an attempt to acquire a reader/writer semaphore in "read" mode when the semaphore is held by a writer will result in the caller pending. Also, if the semaphore is held by other readers but the maximum concurrent readers has been reached the caller will pend. If a task has attempted to take the semaphore in "write" mode and pended for any reason all subsequent "read" takes will result in the caller pending until all writers have run.

When a reader/writer semaphore becomes available a new owner is selected from any tasks pended on the semaphore. If tasks are pended in "write" mode they will be granted ownership in the order determined by the option specified for the semaphore at creation (**SEM_Q_FIFO** or **SEM_Q_PRIORITY**). If no write tasks are pended then all tasks waiting for the semaphore in "read" mode, up to the maximum concurrent readers specified for the semaphore, will be granted ownership in "read" mode.

Though the maximum number of concurrent readers is set per semaphore at creation there is also a limit on the maximum concurrent readers for a system as defined by **SEM_RW_MAX_CONCURRENT_READERS**. The value of **SEM_RW_MAX_CONCURRENT_READERS** will be used as the semaphore's maximum if a larger value is specified at creation. This value should be set no larger than necessary as a larger maximum concurrent reader value will result in longer interrupt and task response.

RECURSIVE RESOURCE ACCESS

Like mutex semaphores reader/writer semaphores support recursive access. Please refer to the **semMLib** documentation for further details.

WARNING

While taking a reader/writer semaphore recursively through either the semWTake and semRTake routines is allowed, an attempt to acquire a semaphore in both modes is not allowed. The **semWTake( )** routine will return **ERROR** if the semaphore is held by the caller as a reader and the **semRTake( )** routine will return **ERROR** if the semaphore is held by the caller as a writer.

PRIORITY-INVERSION SAFETY

Like mutex semaphores reader/writer semaphores support priority inheritence. Please refer to the **semMLib** documentation for further details.

SEMAPHORE DELETION

The **semDelete( )** call terminates a semaphore and deallocates any associated memory. The deletion of a semaphore unblocks tasks pended on that semaphore; the routines which were pended return **ERROR**. Take special care when deleting read/write semaphores to avoid deleting a semaphore out from under tasks that have taken that semaphore. In particular, a semaphore should never be deleted when held in read mode and the option **SEM_DELETE_SAFE** was passed at creation.

Applications should adopt the protocol of only deleting semaphores that the deleting task owns in write mode.

TASK-DELETION SAFETY

Like mutex semaphores reader/writer semaphores support task deletion safety. Please refer to the **semMLib** documentation for further details.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semMLib**, **semBLib**, **semCLib**, VxWorks Programmer's Guide

setenv

NAME

setenv – POSIX environment variable **setenv( )** and **unsetenv( )** routines

ROUTINES

setenv( ) – add or change an environment variable (POSIX)
unsetenv( ) – remove an environment variable (POSIX)
putenv( ) – change or add a value to the environment

DESCRIPTION

This module contains the POSIX compliant **setenv( )** and **unsetenv( )** routines to add, change or remove environment variables from the RTP's environment.

Although these routines are thread-safe, if the application directly modifies the *environ* array or the pointer to which it points, the behavior of **setenv( )** and **unsetenv( )** is undefined.

These routines may not be used in constructors. Doing so would result in the interruption of the execution of the application.

INCLUDE FILES

stdlib.h

---

shmLib

NAME

shmLib – POSIX shared memory objects

ROUTINES

shm_open( ) – open a shared memory object
shm_unlink( ) – remove a shared memory object

DESCRIPTION

This library provides interface for opening, creating and unlinking POSIX shared memory objects.

The shared memory object name space is managed with the help of a special pseudo-file system, shmFs, that has no storage. It provides the functionality to create, open and control shared memory objects in a flat directory structure (all objects are in the shmFs root). Since there is no storage associated to shared memory objects, file read and write are not supported. Instead, they can only be accessed by memory-mapping them (see **mmanLib**), and perfoming direct memory access.

Support for shared memory objects is included in the system via the **INCLUDE_POSIX_SHM** component. When this component is included, the shmFs is automatically initialized and mounted as "/shm". The name of the file system can be changed via the **SHM_DEV_NAME** configuration parameter.

A file descriptor opened for a shared memory object can be probed with the **S_TYPEISSHM( )**, which takes a pointer to a "struct stat" as input.

A shared memory object is completely removed from a system when all of the following conditions are satisfied:

1. It has been unlinked with **shm_unlink( )**. This removes the object from the name space, but the object is not deleted unless the other two conditions are also satified.
2. All file descriptors opened for the object are closed. Note that when a process exits, all open file descriptors are closed automatically.

3. All processes completely unmapped it. Note that when a process exits, all mapped objects are unmapped automatically.

All memory (virtual and physical) once mapped with the **MAP_SHARED** option for a shared memory object remains reserved until the shared memory object is completely removed from the system under the above conditions.

GETTING INFO

The shared memory object file system supports many commonly used standard IO routines, such as **fstat( )**, **ioctl( )**, **fcntl( )**, **pathconf( )**, **ftruncate( )**, when applicable. The content of the file system can also be listed with the **ls( )** and **ll( )** file systems utilities.

Information about mappings of shared memory objects can be obtained with the **mmapShow( )** and the **rtpMemShow( )** kernel shell show routines.

EXAMPLE

The typical usage of shared memory objects is shown in the following example, with a producer and a consumer process:

```
/*
 * Producer process: fill object with some data.
 */

#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

int main ()
    {
    int   fd;
    int   ix;
    int * pData;

    /* create a new SHM object */

    fd = shm_open("/myshm", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1)
        exit (1);

    /* set object size */

    if (ftruncate (fd, 0x1000) == -1)
        exit (1);

    /* Map shared memory object in the address space of the process */

    pData = (int *) mmap (0, 0x1000,  PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);

    if (pData == (int *) MAP_FAILED)
        exit (1);

    /* close the file descriptor; the mapping is not impacted by this */
```

```
    close (fd);

    /* The mapped image can now be written via the pData pointer */

    for (ix = 0; ix < 25; ix++)
        *(pData + ix) = ix;

    /* when the process exits, the object is automatically unmapped */

    exit (0);
    }

/*
 * Consumer process: read object data.
 */

#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>

int main ()
    {
    int   fd;
    int   ix;
    int * pData;

    /* open the SHM object; this must already be created by the producer */

    fd = shm_open("/myshm", O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1)
        exit (1);

    /* Map shared memory object in the address space of the process */

    pData = (int *) mmap (0, 0x1000,  PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);

    if (pData == (int *) MAP_FAILED)
        exit (1);

    /* close the file descriptor; the mapping is not impacted by this */

    close (fd);

    /* The mapped image can now be accessed via the pData pointer */

    for (ix = 0; ix < 25; ix++)
        printf ("%d\n", *(pData + ix));

    /* unlink object (delete from shmFs) */
```

```
shm_unlink ("/myshm");

/* when the process exits, the object is automatically unmapped */

exit (0);
}
```

INCLUDE FILES **sys/mman.h**

SEE ALSO POSIX 1003.1, **mmanLib**

sigLib

NAME **sigLib** – user signal facility library

ROUTINES

sigvec( ) – install a signal handler
sigsetmask( ) – set the signal mask
sigblock( ) – add to a set of blocked signals
sigemptyset( ) – initialize a signal set with no signals included (POSIX)
raise( ) – send a signal to the calling RTP (POSIX)
rtpRaise( ) – send a signal to the calling RTP
taskRaise( ) – send a signal to the calling task
sigfillset( ) – initialize a signal set with all signals included (POSIX)
sigaddset( ) – add a signal to a signal set (POSIX)
sigdelset( ) – delete a signal from a signal set
sigismember( ) – test to see if a signal is in a signal set (POSIX)
signal( ) – specify the handler associated with a signal (POSIX)
sigaction( ) – examine and/or specify the action associated with a signal (POSIX)
wait( ) – wait for any child RTP to terminate (POSIX)
sigwaitinfo( ) – wait for signals (POSIX)
sigwait( ) – wait for a signal to be delivered (POSIX)
sigqueue( ) – send a queued signal to a RTP (POSIX)
rtpSigqueue( ) – send a queued signal to a RTP
_rtpSigqueue( ) – send a queued signal to an RTP with a specific signal code (syscall)
taskSigqueue( ) – send a queued signal to a RTP task
sigprocmask( ) – examine and/or change the signal mask for an RTP (syscall)
sigaltstack( ) – set or get signal alternate stack context (syscall)
sigpending( ) – retrieve the set of pending signals (syscall)
sigsuspend( ) – suspend the task until delivery of a signal
pause( ) – suspend the task until delivery of a signal
kill( ) – send a signal to an RTP (syscall)
rtpKill( ) – send a signal to a RTP

taskKill( ) – send a signal to a RTP task (syscall)
sigtimedwait( ) – wait for a signal
_taskSigqueue( ) – send queued signal to an RTP task with specific signal code (syscall)
_sigqueue( ) – send a queued signal to a RTP with a specific signal code (syscall)

DESCRIPTION

This library provides the signal interfaces in the RTP environment. The signal model in user-mode is designed to follow the POSIX process model.

Signals alter the flow of control of tasks by communicating asynchronous events within or between task contexts. Using the API's provided by this library, signals may be sent from an RTP task to either another RTP or a public task in another RTP.

Signals can be sent to an RTP using the **kill( )** or **raise( )** functions, and will be caught by any task in that RTP which has unmasked that signal. Signals may also be sent to specific task's in the current or another RTP using the **taskKill( )** function.

Tasks that receive signals may either be waiting synchronously for the signal, or may have their signal mask setup to unblock that signal. If there is no such task waiting for the signal, the signal remains pended in the RTP and will be delivered when one such task becomes available.

Users can register signal handlers for each signal. These signal handlers are applicable to the whole RTP, and are not specific to any one task in that RTP. However, a signal mask is associated with each task. When a task is created, its signal mask is inherited from the task that created it. If the parent is a kernel task (e.g. an RTP spawned from the kernel), the signal mask is intialized such that all signals are unblocked.

The following are the default signal actions for the various signals:

1) STOP signals (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
   The RTP is stopped. In other words, all tasks within the RTP are put into the **WIND_STOP** state.

2) SIGCONT signal
   RTP is continued. All tasks belonging to the RTP are brought out of the **WIND_STOP** state.

3) SIGCHLD signal
   This signal is ignored.

4) All other signals
   The actual behaviour depends on the ED&R policy that has been set. A fatal RTP ED&R event is thrown. In most cases this results in termination of the RTP, but this default can be changed.

When an RTP task generates an exception, a signal is sent to that task. This results in the injection of a fatal RTP ED&R event, which in turn results in RTP termination. Note that the signal number chosen to indicate an exception is architecture dependent, but mapped according to what the POSIX standard specifies. The task should have that signal number unmasked if it wishes to receive exception notification.

Signals sent to tasks that are blocked in the kernel are processed as follows:

1. If the task is blocked on an interruptible object in the kernel, it is unblocked and the system call returns **ERROR** with errno set to **EINTR**.

   Object types that can be created from RTP's and made interruptible are:

   Semaphores (Binary, Counting and Mutex), Message Queues, POSIX Semaphores and POSIX Message Queues.

   Semaphores are made interruptible by using the option **SEM_INTERRUPTIBLE** when they are created. Similarly message queues are made interruptible by using the option **MSG_Q_INTERRUPTIBLE** when they are created. See the **semLib** and **msgQLib** documentation for more details.

2. If the task is blocked on a non-interruptible object or resource, signal delivery is postponed until it returns from the system call.

The list of signals and their associated signal numbers is given in **signal.h**.

INCLUDE FILES
signal.h

SEE ALSO
rtpLib, **edrLib**, **taskLib**, **semLib**, **msgQLib**, *Posix 1003.1 specification 2004 edition* (**http://www.opengroup.org)**.

snsLib

NAME
snsLib – Socket Name Service library

ROUTINES

DESCRIPTION
This library implements the Socket Name Service (SNS). SNS allows applications based on the Socket Application Library (SAL) to associate a list of socket addresses with a service name. This name can then be referenced by other SAL-based applications to determine which socket addresses the server application providing the specified service is using.

SERVICE INFORMATION
SNS maintains a simple database of service entries. Each service entry contains the following information:

Service Name:
A character string mnemonic for the service.

Service Scope:
Level of visibility of the service within the system.

Service Sockets:
Information about the sockets which provide the service.

Service Owner:
The entity that created the service (operating system kernel or RTP identifier).

SERVICE LOCATOR

An application that wishes to register a new service, or locate an existing service, must specify a "service location". The service location is simply the service's name, optionally attached with a scope indicator in URL format. All locations must be unique within a scope.

SERVICE SCOPE

The service scoping capability of SNS allows a server application to limit the visibility of a service name to a specified subset of applications within a system. An analogous capability allows a client application searching for a specified service name to limit how far SNS should search. Thus, a search only returns a matching entry if the search scope specified by the client overlaps the service scope specified by the server. Four levels of scope are supported:

private:
The service is visible within the service's memory region (the operating system kernel space or RTP) only.

node:
The service is visible within the service's owner local node only.

cluster:
The service is visible within the service's cluster of nodes only.

system:
The service is visible to all nodes in the system.

The scoping capability of SNS is best illustrated by visualizing the SNS name space as a set of nested boxes, each representing a different scope.

SNS currently supports the exchange of service information between nodes using the TIPC (Transparent Inter-Process Communication) protocol. Thus the TIPC component must be included in a project to utilize the distributed mode of operation. Services with a scope of the "system" and "cluster" will be visible to an application on another node (if the address family allows it).

It is possible to create **AF_LOCAL** sockets with scope larger than the node, but these sockets will not be visible outside of the node on which they were created.

URL SCHEME SYNOPSIS

[SNS:]service_name[@scope]

where the parts in brackets, [ ], are optional.

SNS: represent the URL service scheme, i.e. the Socket Name Service. It is the only scheme accepted and can be omitted.

@scope represents the visibility of the service name within the system. It can take several values, depending from the context and the application needs. If the the scope is not specified, "@node" is assumed.

The URL representation is case insensitive.

For SAL service creation, registration or removal **service_name** cannot contain any wildcard symbol, and scope must be the exact scope such as **node**, **private**, **cluster** and **system**. **service_name** should not contain RFC 2396 reserved characters.

For the SAL client (to open or find services), **service_name** make contain wildcard, and scope may provide exact scope or the outmost scope. For detail, refer to the **SERVICE DISCOVERY** section below.

SERVICE REGISTRATION AND DISCOVERY

A service who wants to take advantage of the SNS capability, registers itself to the system by providing an URL format identifier. If no scope is specified, the default is set to **node**.

A client discovers a server by specifying the service URL. In this case, there are two search options for each level of visibility:

- if a user specifies the service URL with the scope as described above, the system looks for the service only within the specified scope.
- if the scope is prefixed with **upto_**, such as **upto_node**, the system searches a service beginning from the **private** scope. If it cannot find one, the search moves outward to the next scope. The search stops either when a service is located or the specified scope has been reached and no service was found.

For example, assuming both client and server are in the same node, if a service is defined with **node** scope and the client specifies a scope **upto_cluster** the search will return the matching service. On the other hand, if the client specifies **cluster** then the search will not return that service. It might still return another service with the same name but in a different node, which registered itself with **cluster** visibility.

CONFIGURATION

Socket Name Service capabilities are provided by an SNS server task, which can be configured to start automatically when VxWorks starts up. The server task can be configured to run in its own RTP or as part of the base operating system.

To use the SNS server, configure VxWorks with either the **INCLUDE_SNS** or the **INCLUDE_SNS_RTP** component. With either component you will also require **INCLUDE_UN_COMP**, **INCLUDE_SAL_COMMON**, and **INCLUDE_SAL_SERVER**.

For the distributed versions of the SNS server, the respective components are **INCLUDE_SNS_MP** and **INCLUDE_SNS_MP_RTP**. Note that an additional task called **dsalMonitor** is started in the kernel to monitor all existing distributed SNS servers in the system.

If the SNS server runs as an RTP, the executable needs to be allocated in the path defined by **SNS_PATHNAME**.

INCLUDE FILES **snsLib.h**

SEE ALSO **salClient**, **salServer**, **snsShow**

strSearchLib

NAME **strSearchLib** – Efficient string search library

ROUTINES
fastStrSearch( ) – Search by optimally choosing the search algorithm
bmsStrSearch( ) – Search using the Boyer-Moore-Sunday (Quick Search) algorithm
bfStrSearch( ) – Search using the Brute Force algorithm

DESCRIPTION This library supplies functions to efficiently find the first occurrence of a string (called a pattern) in a text buffer. Neither the pattern nor the text buffer needs to be null-terminated.

The functions in this library search the text buffer using a "sliding window" whose length equals the pattern length. First the left end of the window is aligned with the beginning of the text buffer, then the window is compared with the pattern. If a match is not found, the window is shifted to the right and the same procedure is repeated until the right end of the window moves past the end of the text buffer.

This library supplies the following search functions:

fastStrSearch( )
Optimally chooses the search algorithm based on the pattern size

bmsStrSearch( )
Uses the efficient Boyer-Moore-Sunday search algorithm; may not be optimal for small patterns

bfStrSearch( )
Uses the simple Brute Force search algorithm; best suited for small patterns

To include this library, configure VxWorks with the **INCLUDE_STRING_SEARCH** component.

INCLUDE FILE **strSearchLib.h**

symLib

NAME **symLib** – symbol table subroutine library

ROUTINES

symTblCreate( ) – create a symbol table
symTblDelete( ) – delete a symbol table
symAdd( ) – create and add a symbol to a symbol table, including a group number
symRemove( ) – remove a symbol from a symbol table
symFindByName( ) – look up a symbol by name
symFindByNameAndType( ) – look up a symbol by name and type
symByValueFind( ) – look up a symbol by value
symByValueAndTypeFind( ) – look up a symbol by value and type
symFindByValue( ) – look up a symbol by value
symFindByValueAndType( ) – look up a symbol by value and type
symEach( ) – call a routine to examine each entry in a symbol table

DESCRIPTION

This library provides facilities for managing symbol tables. A symbol table associates a name and type with a value. A name is simply an arbitrary, null-terminated string. A symbol type is an unsigned char (typedef **SYM_TYPE**). A symbol value is a pointer. Though commonly used as the basis for object loaders, symbol tables may be used whenever efficient association of a value with a name is needed.

If you use the **symLib** subroutines to manage symbol tables local to your own applications, the values for **SYM_TYPE** objects are completely arbitrary; you can use whatever one-byte integers are appropriate for your application.

USAGE

Tables are created with **symTblCreate( )**, which returns a symbol table ID. This ID is used for all symbol table operations, including adding symbols, removing symbols, and searching for symbols. All operations on a symbol table are protected from re-entrancy problems by means of a mutual-exclusion semaphore in the symbol table structure. To ensure proper use of the symbol table semaphore, all symbol table accesses and operations should be performed using the API's provided by the **symLib** library. Symbol tables are deleted with **symTblDelete( )**.

Symbols are added to a symbol table with **symAdd( )**. Each symbol in the symbol table has a name, a value, a type and a reference. Symbols are removed from a symbol table with **symRemove( )**.

Symbols can be accessed by either name or value. The routine **symFindByName( )** searches the symbol table for a symbol with a specified name. The routine **symByValueFind( )** finds a symbol with a specified value or, if there is no symbol with the same value, the symbol in the table with the largest value that is smaller than the specified value. Using this method, if an address is inside a function whose name is registered as a symbol, then the name of the function will be returned.

The routines **symFindByValue( )** and **symFindByValueAndType( )** are obsolete. They are replaced by the routines **symByValueFind( )** and **symByValueAndTypeFind( )** and will be removed in the next version of VxWorks.

Symbols in the symbol table are hashed by name into a hash table for fast look-up by name, e.g., by **symFindByName( )**. The size of the hash table is specified during the creation of a symbol table. Look-ups by value, e.g., **symByValueFind( )**, must search the table linearly; these look-ups can therefore be much slower.

The routine **symEach( )** allows every symbol in the symbol table to be examined by a user-specified function.

Name clashes occur when a symbol added to a table is identical in name and type to a previously added symbol. Whether or not symbol tables can accept name clashes is set by a parameter when the symbol table is created with **symTblCreate( )**.

If name clashes are not allowed, **symAdd( )** will return an error if there is an attempt to add a symbol with the same name and type as a symbol already in the symbol table.

If name clashes are allowed, adding multiple symbols with the same name and type will be permitted. In such cases, **symFindByName( )** will return the value most recently added, although all versions of the symbol can be found using **symEach( )**.

INCLUDE FILES **symLib.h**

ERRNOS Routines from this library can return the following symbol-specific errors:

S_symLib_SYMBOL_NOT_FOUND
The requested symbol can not be found in the specified symbol table.

S_symLib_NAME_CLASH
A symbol of same name already exists in the specified symbol table (only when the name clash policy is selected at symbol table creation).

S_symLib_TABLE_NOT_EMPTY
The symbol table is not empty from its symbols, and then can not be deleted.

S_symLib_INVALID_SYMTAB_ID
The symbol table ID is invalid.

S_symLib_INVALID_SYM_ID_PTR.
The symbol table ID pointer is invalid.

S_symLib_INVALID_SYMBOL_NAME
The symbol name is invalid.

Note that other errors, not listed here, may come from libraries internally used by this library.

SEE ALSO **loadLib**

sysLib

NAME **sysLib** – system dependent APIs

ROUTINES

syscallInfo( ) – get information on a system call from user mode
syscallPresent( ) – check if a system call is present from user mode
syscallGroupPresent( ) – check if a system call group is present from user mode
syscallNumArgsGet( ) – return the number of arguments a system call takes
syscallGroupNumRtnGet( ) – return the number of routines in a system call group
sysClkRateGet( ) – get the system clock rate
sysAuxClkRateGet( ) – get the auxilary clock rate
sysProcNumGet( ) – get the processor number
sysBspRev( ) – get the BSP version and revision number
sysModel( ) – get the model name of the CPU board
sysMemTop( ) – get the address of the top of logical memory
sysPhysMemTop( ) – get the address of the top of physical memory

DESCRIPTION

This library contains various system APIs that describe the state of the running VxWorks system. In kernel mode, many of these APIs are provided by the BSP.

In user mode, most of these APIs obtain their results by sending a sysctl request to the kernel.

Some APIs are implemented as macros.

CONFIGURATION

The system dependent APIs are automatically included for use in an RTP when RTPs are enabled with **INCLUDE_RTP**. (Note that removing **INCLUDE_SYSCTL** results in an error when these APIs are called.)

INCLUDE FILES

sysLib.h

sysconf

NAME

sysconf – POSIX 1003.1/1003.13 (PSE52) **sysconf( )** API

ROUTINES

sysconf( ) – get configurable system variables

DESCRIPTION

This module contains the POSIX conforming **sysconf( )** routine used by applications to get the values of "configurable system variables". Configurable system variables represent either limits or features that this implementation of POSIX 1003.1 supports, in agreement with the PSE52 profile of the IEEE 1003.13 standard.

INCLUDE FILES

unistd.h, **limits.h**

sysctlLib

NAME **sysctlLib** – sysctl userland routines

ROUTINES **sysctl( )** – get or set the the values of objects in the sysctl tree (syscall)

DESCRIPTION This module documents the use of the sysctl system call from user space.

The following sysctl variables can be read from user mode (this is not an exhaustive list, but these are actively supported. Use **Sysctl "-A"** from the VxWorks shell to get the list of variables currently active on the system).

KERN.OSTYPE
  string: system's type (i.e. OS name).

KERN.OSRELEASE
  string: system's full release number (i.e. major.minor.maintenance).

KERN.OSREV
  string: system's version information (reserved for future use).

KERN.OSBUILDDATE
  string: system's build date.

KERN.VERSION
  string: kernel's version string.

KERN.TICKGET
  long: current tick count

KERN.TICK64GET
  long long: current tick count (64 bit)

KERN.SYSCALL
  node: information on system calls and system call groups

KERN.SYSCALL.*syscallNum* | *groupNum*.NAME
  string: system call name

KERN.SYSCALL.*syscallNum* | *groupNum*.NARGS
  int: number of arguments taken

KERN.SYSCALL.*syscallNum* | *groupNum*.GROUP
  string: system call group name

KERN.SYSCALL.*syscallNum* | *groupNum*.GROUP_NROUTINE
  int: number of routines in group

CONFIGURATION To use the sysctl system call from user mode, configure VxWorks with the **INCLUDE_SC_SYSCTL** component; *errno* is set to **ENOSYS** if this component is not included. (This component is automatically included when **INCLUDE_RTP** is configured.)

INCLUDE FILES **sys/sysctl.h**

taskHookLib

NAME **taskHookLib** – user-level task hook library

ROUTINES

taskCreateHookAdd( ) – add a routine to be called at every task create
taskCreateHookDelete( ) – delete a previously added task create routine
taskDeleteHookAdd( ) – add a routine to be called at every task delete
taskDeleteHookDelete( ) – delete a previously added task delete routine

DESCRIPTION

This library provides routines for adding extensions to the VxWorks user-level tasking facility. To allow task-related facilities to be added to the system without modifying the task support library, the library provides call-outs every time a task is created or deleted. The call-outs allow additional routines, or "hooks," to be invoked whenever these events occur. The hook management routines below allow hooks to be dynamically added to and deleted from the current lists of create and delete hooks:

taskCreateHookAdd( ) and **taskCreateHookDelete( )**
Add and delete routines to be called when a task is created.

taskDeleteHookAdd( ) and **taskDeleteHookDelete( )**
Add and delete routines to be called when a task is deleted.

NOTE

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

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

EXAMPLE

In the following code example, a customer library installs a task create hook and a task delete hook. The create hook attaches an application specific structure to each created task.

```
/* locals */

static TLS_KEY appKey = 0;
```

```
/* forward declarations */

extern STATUS appTaskCreateHook (int tid);
extern STATUS appTaskDeleteHook (int tid);


_WRS_CONSTRUCTOR (appLibInit, xx)
    {

    /* allocate a slot in the thread-local storage (TLS) area */

    appKey = tlsKeyCreate ();

    /* register task create and delete hooks */

    if (taskCreateHookAdd ((FUNCPTR) appTaskCreateHook) == ERROR)
        {
        exit (<insert error code>);
        }

    if (taskDeleteHookAdd ((FUNCPTR) appTaskDeleteHook) == ERROR)
        {
        exit (<insert error code>);
        }

    /* other initialization */

    ...
    }


STATUS appTaskCreateHook
    (
    int tid
    )
    {
    APP_STRUCT *myStruct;  /* ptr to application specific per-task struct */

    /* allocate application specific per-task struct */

    if (myStruct = (APP_STRUCT *) malloc (sizeof (APP_STRUCT)) == NULL)
        {
        return (ERROR);
        }

    /* initialize application specific per-task struct */

    bzero ((char *) myStruct, sizeof (APP_STRUCT));

    /* register ptr to application specific per-task struct in TLS */

    if (tlsValueOfTaskSet (tid, appKey, (void *) myStruct) == ERROR)
        {
        free ((char *) myStruct);
        return (ERROR);
        }
```

```
    return (OK);
    }

STATUS appTaskDeleteHook
   (
   int tid
   )
   {
   STATUS      status;
   APP_STRUCT *myStruct;  /* ptr to application specific per-task struct */

   /* obtain ptr to application specific per-task struct */

   status = tlsValueOfTaskGet (tid, appKey, (void **) &myStruct);

   if ((status == ERROR) || (myStruct == NULL))
       {
       return (ERROR);
       }

   /* perform application specific tear down actions */

   ...

   /* free application specific per-task struct */

   free ((char *) myStruct);

   return (OK);
   }

STATUS appAction
    (
    int tid,            /* task to perform action against */
    <other arguments>
    )
    {
    STATUS      status;
    APP_STRUCT *myStruct; /* ptr to application specific per-task struct */

    /* obtain ptr to application specific per-task struct */

    status = tlsValueOfTaskGet (tid, appKey, (void **) &myStruct);

    if ((status == ERROR) || (myStruct == NULL))
       {
       errno = S_objLib_OBJ_ID_ERROR;  /* or some other errno value */
       return (ERROR);
       }

    /* free application specific per-task action */

    ...

    return (OK);
```

```
}
```

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**

taskInfo

NAME **taskInfo** – task information library

ROUTINES

taskName( ) – get the name of a task residing in the current RTP
taskNameGet( ) – get the name of any task
taskInfoGet( ) – get information about a task
taskOptionsGet( ) – examine task options
taskNameToId( ) – look up the task ID associated with a task name
taskIdDefault( ) – set the default task ID
taskIsReady( ) – check if a task is ready to run
taskIsSuspended( ) – check if a task is suspended
taskIsPended( ) – check if a task is pended

DESCRIPTION This library provides a programmatic interface for obtaining task information.

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines **taskOptionsGet( )**, **taskName( )**, **taskNameGet( )**, **taskNameToId( )**, **taskIsReady( )**, **taskIsPended( )**, and **taskIsSuspended( )** are used to obtain task information.

The chief drawback of using task information is that tasks may change their state between the time the information is gathered and the time it is utilized. Information provided by these routines should therefore be viewed as a snapshot of the system, and not relied upon unless the task is consigned to a known state, such as suspended.

Task management and control routines are provided by **taskLib**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **semLib**

taskLib

NAME

taskLib – VxWorks user task-management library

ROUTINES

taskOpen( ) – open a task
taskClose( ) – close a task
taskUnlink( ) – unlink a task
taskSpawn( ) – spawn a task
taskCreate( ) – allocate and initialize a task without activation
taskExit( ) – exit a task
taskActivate( ) – activate a task that has been created without activation
taskDelete( ) – delete a task
taskDeleteForce( ) – delete a task without restriction
taskSuspend( ) – suspend a task
taskResume( ) – resume a task
taskRestart( ) – restart a task
taskPrioritySet( ) – change the priority of a task
taskPriorityGet( ) – examine the priority of a task
taskPriNormalGet( ) – examine the normal priority of a task
taskRtpLock( ) – disable task rescheduling
taskRtpUnlock( ) – enable task rescheduling
taskSafe( ) – make the calling task safe from deletion
taskUnsafe( ) – make the calling task unsafe from deletion
taskIdSelf( ) – get the task ID of a running task
taskIdVerify( ) – verify the existence of a task
_taskOpen( ) – open a task (system call)
taskDelay( ) – delay calling task from executing (system call)
taskCtl( ) – perform a control operation against a task (system call)

DESCRIPTION

This library provides the interface to the VxWorks user task management facilities. Task control services are provided by **taskLib**, **semLib**, and **msgQLib**. Programmatic access to task information and debugging features is provided by **taskInfo**.

Although of the tasking services are provided by the VxWorks kernel, several primitives contain a significant level of implementation in user-mode to minimize the occurance of system calls. For example, the task-deletion-safe primitives and preemption-lock primitives typically do not result in any system calls being issued, and thus provide performance comparable to the kernel.

TASK CREATION

Application tasks are typically created with the routines **taskSpawn( )**, **taskCreate( )**, or **taskOpen( )**.

The **taskOpen( )** API is the most general-purpose task-creation routine because it accepts a *mode* parameter to control various object-management-related options, and also allows creating a task as **public** or **private**. A **public** object is visible to all RTPs in the system, whereas a **private** object is only visible to the RTP in which the object resides.

The **VX_TASK_NOACTIVATE** option bit prevents the task from being activated upon creation. Without this option the task is activated. The **taskOpen( )** API also permits the location of the user stack area to be specified.

In addition to creating tasks, the **taskOpen( )** routine can be used to obtain a handle to an already existing task. Typically this is used to obtain a task identifier of a public task in another RTP.

The **taskSpawn( )** and **taskCreate( )** routines, which are similar to VxWorks 5.x routines, create **private** tasks only. The two functions are similiar, except that **taskCreate( )** does *not* activate the task upon creation. As when the **VX_TASK_NOACTIVATE** option bit is specified with **taskOpen( )**, a subsequent call to **taskActivate( )** is required to activate the task.

Application tasks execute in the least privileged state of the underlying architecture, which means in user mode as opposed to supervisor mode. Thus certain operations are off limits to application tasks, including executing privileged instructions and accessing hardware registers.

There is no limit to the number of tasks that can be created in an RTP, if sufficient memory is available in the RTP and kernel heaps to satisfy allocation requirements.

Application routines can be hooked into the task creation mechanism using **taskHookCreateAdd( )**.

TASK DELETION

If a task exits its **main** routine, specified during task creation, this library implicitly calls **taskExit( )** to delete the task. Tasks can be explicitly deleted with the **taskDelete( )** or **taskExit( )** routines. The **exit( )** function differs from **taskExit( )** in that invoking the **exit( )** routine causes the entire RTP to terminate.

Task deletion must be handled with extreme care, due to the inherent difficulties of resource reclamation. Deleting a task that owns a critical resource can cripple the system, since the resource may no longer be available. Simply returning a resource to an available state is not a viable solution, since the system can make no assumption as to the state of a particular resource at the time a task is deleted.

The solution to the task deletion problem lies in deletion protection, rather than overly complex deletion facilities. Tasks may be protected from unexpected deletion using **taskSafe( )** and **taskUnsafe( )**. While a task is safe from deletion, deleters will block until it is safe to proceed. Also, a task can protect itself from deletion by taking a mutual-exclusion semaphore created with the **SEM_DELETE_SAFE** option, which enables an implicit **taskSafe( )** with each **semTake( )**, and a **taskUnsafe( )** with each **semGive( )**. (For more information, see **semLib**.) Many VxWorks library resources are protected in this manner, and application designers may wish to consider this facility in cases where dynamic task deletion is a possibility.

A task cannot delete another task unless they both reside in the same RTP.

The **rtpSigLib** facility may be used to allow a task to execute clean-up code before actually expiring. Application routines can be hooked into the task deletion mechanism using **taskHookDeleteAdd( )**.

TASK CONTROL Tasks are manipulated by means of an ID that is returned when a task is created. VxWorks uses the convention that specifying a task ID of **NULL** in a task control function signifies the calling task.

The following routines control task state: **taskResume( )**, **taskSuspend( )**, **taskDelay( )**, **taskRestart( )**, and **taskPrioritySet( )**.

TASK SCHEDULING

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.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskInfo**, **taskHookLib**, **semLib**

taskUtilLib

NAME **taskUtilLib** – task utility library

ROUTINES
taskCpuAffinitySet( ) – set the CPU affinity of a task
taskCpuAffinityGet( ) – get the CPU affinity of a task

DESCRIPTION This library provides a programmatic interface for obtaining and modifying task information.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**

tickLib

NAME **tickLib** – tick routines

ROUTINES

tickGet( ) – get the value of the kernel's tick counter
tick64Get( ) – get the value of the kernel's tick counter as a 64 bit value

DESCRIPTION

This library contains miscellaneous kernel routines, namely **tickGet( )**, **tick64Get( )** and **sysClkRateGet( )**.

INCLUDE FILES

tickLib.h

timerLib

NAME

timerLib – user-level timer library (POSIX)

ROUTINES

timer_cancel( ) – cancel a timer
timer_connect( ) – connect a user routine to the timer signal
timer_create( ) – allocate a timer using the specified clock for a timing base (POSIX)
timer_open( ) – open a timer
timer_close( ) – close a named timer
timer_unlink( ) – unlink a named timer
timer_delete( ) – remove a previously created timer (POSIX)
timer_gettime( ) – get the remaining time before expiration and the reload value (POSIX)
timer_getoverrun( ) – return the timer expiration overrun (POSIX)
timer_settime( ) – set the time until the next expiration and arm timer (POSIX)
nanosleep( ) – suspend the current task until the time interval elapses (POSIX)
sleep( ) – delay for a specified amount of time
alarm( ) – set an alarm clock for delivery of a signal
_timer_open( ) – open a kernel POSIX timer (system call)
timer_ctl( ) – performs a control operation on a kernel timer (system call)

DESCRIPTION

This library provides a timer interface, as defined in the IEEE standard, POSIX 1003.1b.

Timers are mechanisms by which process or task signal themselves after a designated interval. Timers are built on top of the clock and signal facilities. The clock facility provides an absolute time-base. Standard timer functions simply consist of creation, deletion and setting of a timer. When a timer expires, **sigaction( )** (see **rtpSigLib**) must be in place in order for the user to handle the event. The "high resolution sleep" facility, **nanosleep( )**, allows sub-second sleeping to the resolution of the clock.

ADDITIONS

Two non-POSIX functions are provided for user convenience:

timer_cancel( )
This routine disables a timer by calling **timer_settime( )**.

timer_connect( )
This routine hooks up a user routine by calling **sigaction( )**. When the timer expires, this routine is called in the context of the task that created the timer.

CLARIFICATIONS
The process or task creating a timer with **timer_create( )** will receive the signal no matter which task actually arms the timer.

As specified by the POSIX standard, the **sleep( )** prototype is defined in **unistd.h**.

IMPLEMENTATION
The actual clock resolution is hardware-specific and in many cases is 1/60th of a second. This is less than **_POSIX_CLOCKRES_MIN**, which is defined as 20 milliseconds (1/50th of a second).

CONFIGURATION
This library requires the **INCLUDE_POSIX_TIMERS** component to be configured into the kernel; *errno* may be set to **ENOSYS** if this component is not present.

The **SIGEV_THREAD** notification type requires POSIX thread support which requires the POSIX Clocks and Pthread Scheduler components (**INCLUDE_POSIX_CLOCKS** and **INCLUDE_POSIX_PTHREAD_SCHEDULER**) to be included in the VxWorks kernel; *errno* may be set to **ENOSYS** if these components have not been configured into the kernel.

INCLUDE FILES
timers.h, **unistd.h**

SEE ALSO
clockLib, **sigaction( )**, POSIX 1003.1b documentation

tlsOldLib

NAME
tlsOldLib – Task Local Storage Library - To be deprecated

ROUTINES
tlsKeyCreate( ) – create a key for the TLS data
tlsValueGet( ) – get a value of a specific TLS data
tlsValueSet( ) – set the value of a TLS data
tlsSizeGet( ) – Get size of the TLS structure

WARNING
This library has been deprecated. The __thread storage class, which supports task local storage, replaces this TLS library.

SMP CONSIDERATIONS
For SMP, this library is not supported. A call to create a TLS key via this library will result in an error and by default, will terminate the RTP. To use task local storage, use the __thread storage class instead.

DESCRIPTION

This library provides the task local storage (TLS) functionality for user mode tasks within the Real Time Process (RTP) space.

For an RTP, the TLS size is fixed once the initial task of the process has completed its initialization. For each task created in the RTP, a TLS is allocated for the new task.

For each TLS, keys are associated with slots in the TLS. The key may be obtain via a call to **tlsKeyCreate( )**. Keys allocated are global to the RTP. Every task uses the same key to access the same slot number in the TLS array. A task can not have its own private key. Once a key has been created, the key stays valid for the lifetime of the RTP.

The current running task in the RTP may access its TLS values using the **tlsValueGet( )** and **tlsValueSet( )** routines.

The maximum number of TLS slots available is the number of **tlsKeyCreate( )** calls plus the additional **tlsAdditionalSlots** (currently set to 10). The tlsAdditionalSlots is defined in **tlsData.c** and can be modified by the user to increase or decrease the number. Modification of this number must be done prior to compiling the RTP executable. Once the maximum number of TLS slots have been allocated, no more TLS keys will be given by the **tlsKeyCreate( )** routine.

INCLUDE FILES

tlsOldLib.h

---

uname

NAME

uname – POSIX 1003.1 **uname( )** API

ROUTINES

uname( ) – get identification information about the system

DESCRIPTION

This module contains the POSIX compliant **uname( )** routine used by applications to get identification information about the system.

INCLUDE FILES

sys/utsname.h

---

usrFsLib

NAME

usrFsLib – file system user interface subroutine library

ROUTINES

cd( ) – change the default directory
pwd( ) – print the current default directory
mkdir( ) – make a directory

rmdir( ) – remove a directory
rm( ) – remove a file
copyStreams( ) – copy from/to specified streams
copy( ) – copy *in* (or stdin) to *out* (or stdout)
chkdsk( ) – perform consistency checking on a MS-DOS file system
dirList( ) – list contents of a directory (multi-purpose)
ls( ) – generate a brief listing of a directory
ll( ) – generate a long listing of directory contents
lsr( ) – list the contents of a directory and any of its subdirectories
llr( ) – do a long listing of directory and all its subdirectories contents
cp( ) – copy file into other file/directory.
mv( ) – mv file into other directory.
xcopy( ) – copy a hierarchy of files with wildcards
xdelete( ) – delete a hierarchy of files with wildcards
attrib( ) – modify MS-DOS file attributes on a file or directory
xattrib( ) – modify MS-DOS file attributes of many files
dosfsDiskFormat( ) – format a disk with dosFs
diskFormat( ) – format a disk with dosFs
hrfsDiskFormat( ) – format a disk with HRFS
diskInit( ) – initialize a file system on a block device
commit( ) – commit current transaction to disk.
ioHelp( ) – print a synopsis of I/O utility functions

DESCRIPTION

This library provides user-level utilities for managing file systems. These utilities may be used from Host Shell, the Kernel Shell or from an application.

USAGE FROM HOST SHELL

Some of the functions in this library have counterparts of the same names built into the Host Shell (aka Windsh). The built-in functions perform similar functions on the Tornado host computer's I/O systems. Hence if one of such functions needs to be executed in order to perform any operation on the Target's I/O system, it must be preceded with an **@** sign, e.g.:

```
-> @ls "/sd0"
```

will list the directory of a disk named "/sd0" on the target, wile

```
-> ls "/tmp"
```

will list the contents of the "/tmp" directory on the host.

The target I/O system and the Host Shell running on the host, each have their own notion of current directory, which are not related, hence

```
-> pwd
```

will display the Host Shell current directory on the host file system, while

```
-> @pwd
```

will display the target's current directory on the target's console.

WILDCARDS
Some of the functions herein support wildcard characters in argument strings where file or directory names are expected. The wildcards are limited to "*" which matches zero or more characters and "?" which matches any single characters. Files or directories with names beginning with a "." are not normally matched with the "*" wildcard.

DIRECTORY LISTING
Directory listing is implemented in one function **dirList( )**, which can be accessed using one of these four front-end functions:

ls( )
produces a short list of files

lsr( )
is like **ls( )** but ascends into subdirectories

ll( )
produces a detailed list of files, with file size, modification date attributes etc.

llr( )
is like **ll( )** but also ascends into subdirectories

All of the directory listing functions accept a name of a directory or a single file to list, or a name which contain wildcards, which will result in listing of all objects that match the wildcard string provided.

INCLUDE FILES
usrLib.h

SEE ALSO
ioLib, **dosFsLib**, **netDrv**, **nfsLib**, **hrFsLib**, the VxWorks programmer guides, the, *VxWorks Command-Line Tools User's Guide*.

---

vxAtomicLib

NAME
vxAtomicLib – atomic operations library

ROUTINES
vxAtomicAdd( ) – atomically add a value to a memory location
vxAtomicAnd( ) – atomically perform a bitwise AND on a memory location
vxAtomicDec( ) – atomically decrement a memory location
vxAtomicInc( ) – atomically increment a memory location
vxAtomicNand( ) – atomically perform a bitwise NAND on a memory location
_vxAtomicOr( ) – atomically perform a bitwise OR on memory location
vxAtomicSub( ) – atomically subtract a value from a memory location
vxAtomicXor( ) – atomically perform a bitwise XOR on a memory location
vxAtomicClear( ) – atomically clear a memory location
vxAtomicGet( ) – atomically get a memory location
vxAtomicSet( ) – atomically set a memory location

vxCas( ) – atomically compare-and-swap the contents of a memory location

DESCRIPTION

This library provides routines to perform a number of atomic operations on a memory location: add, subtract, increment, decrement, bitwise OR, b   itwise NOR, bitwise AND, bitwise NAND, set, clear and compare-and-swap.

Atomic operations constitute one of the solutions to the mutual exclusion problems faced by multi-threaded applications. The ability to perform an indivisible read-modify-write operation on a memory location allows multiple threads of execution or tasks to safely read-modify-write a global variable. Mutex semaphores, and task locking are other mutual exclusion mechanisms that exist in VxWorks.

INCLUDE FILES

vxAtomicLib.h

vxCpuLib

NAME

vxCpuLib – CPU utility routines

ROUTINES

vxCpuEnabledGet( ) – get a set of running CPUs
vxCpuConfiguredGet( ) – get the number of configured CPUs in the system

DESCRIPTION

This library provides a small number of utility routines for users who need to have visibility into the number of CPUs that are present in a VxWorks system.

Routines in this library allow a user to determine:

- The number of CPUs configured in the system.
- The number of enabled CPUs in the system.

CPU SETS

Routine **vxCpuEnabledGet( )** returns the set of enabled CPUs in the system. In VxWorks a set of CPUs is always represented using a cpuset_t type variable. Refer to the reference entry for **cpuset** to obtain more information.

INCLUDE FILES

vxCpuLib.h

SEE ALSO

cpuset, The VxWorks Programmer's Guides

wvScLib

NAME

wvScLib – System calls for System Viewer

ROUTINES **wvEvent( )** – record a System Viewer user event

DESCRIPTION This library contains the system calls related to the Wind River System Viewer in VxWorks. It requires the System Viewer components to be included.

INCLUDE FILES

2

Routines

2

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_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_ISSET**

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.

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**

_edrErrorInject( )

NAME **_edrErrorInject( )** – inject an ED&R error record (system call)

SYNOPSIS

```
STATUS _edrErrorInject
    (
    int              kind,
    const char      *fileName,
    int              lineNumber,
    REG_SET         *regset,
    void            *addr,
    const char      *msg
    )
```

DESCRIPTION This syscall 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_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.

RETURNS

OK if the error was stored correctly, or **ERROR** if some failure occurs during storage

ERRNO

S_edrLib_NOT_INITIALIZED
The ED&R library was not initialized

S_edrLib_PROTECTION_FAILURE
The ED&R memory log could not be protected or unprotected

S_edrLib_INVALID_OPTION
An invalid facility or severity was provided

SEE ALSO

edrLib

_exit( )

NAME

_exit( ) – terminate the calling process (RTP) (syscall)

SYNOPSIS

```
void _exit
    (
    int status
    )
```

DESCRIPTION

This routine terminates the calling RTP. All open file descriptors in the calling process are closed. Memory allocated to the RTP will also be freed back to the system. Any RTP delete hooks installed in the kernel will execute before the deletion is performed.

Most C programs should call the library routine **exit( )** to terminate the process. Calling **exit( )** invokes routines registered via the **atexit( )** routine. Calling the **_exit( )** function directly skips the calls to atexit routines, and is therefore considered an abnormal termination of the calling process.

RETURNS

N/A.

ERRNO

SEE ALSO

rtpLib, **_Exit( )**, **exit( )**, **rtpExit( )**, the VxWorks programmer guides

_getcwd( )

NAME

_getcwd( ) – get pathname of current working directory (syscall)

SYNOPSIS

```
char * _getcwd
    (
    char * buffer,
    size_t length
    )
```

DESCRIPTION

The **_getcwd( )** function copies the current working directory pathname into the user provided buffer. The value of *length* must be at least one greater than the length of the pathname to be returned.

The regular **getcwd( )** function maps the **ERROR** return value to a **NULL** value to follow the standard API.

RETURNS

pointer to user buffer, or **ERROR**.

ERRNO

EINVAL
invalid arguments.

ERANGE
Buffer is not large enough to receive the path name.

SEE ALSO

ioLib, **getcwd( )**

_mctl( )

NAME

_mctl( ) – invoke memory control functions (syscall)

SYNOPSIS

```
int _mctl
   (
   void *    addr,              /* address to memory block */
   size_t    len,               /* size of memory block */
   int       func,              /* control function number to invoke */
   int       arg                /* argument for control function */
   )
```

DESCRIPTION

This routine invokes one of the predefined memory control functions. It should be invoked via the wrapper functions, as listed below:

Function	Argument	Comments
MCTL_CACHE_FLUSH	cache type	Use **cacheFlush( )** instead.
MCTL_CACHE_INVALIDATE	cache type	Use **cacheInvalidate( )** instead.
MCTL_CACHE_CLEAR	cache type	Use **cacheClear( )** instead.
MCTL_CACHE_TEXT_UPDATE	unused	Use **cacheTextUpdate( )** instead.
MCTL_MSYNC	msync flag	Use **msync( )** instead.
MCTL_MPROBE	protection	Use **mprobe( )** instead.

RETURNS

0 on success, or -1 in case of failure.

ERRNO

ENOTSUP
Control function not supported.

ENOMEM
Some or all of the address range specified by the addr and len arguments do not correspond to valid mapped pages in the address space of the process.

EINVAL
Invalid argument passed to control function.

SEE ALSO

mmanLib, **cacheLib**

_msgQOpen( )

NAME

_msgQOpen( ) – open a message queue (system call)

SYNOPSIS

```
MSG_Q_ID msgQOpen
    (
    const char * name,          /* message queue name */
    UINT           maxMsgs,             /* max messages that can be queued */
    UINT           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 it searchs the name space and returns the **MSG_Q_ID** of an existing message queue with *name*. If none is found, it creates a new message queue with *name* according to the flags set in the *mode* parameter.

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.

A description of the *mode* and *context* arguments follows. See the reference entry for **msgQCreate( )** for a description of the remaining arguments.

mode

This parameter specifies the various object management attribute bits as follows:

OM_CREATE

Create a new message queue if a matching message queue name is not found.

OM_EXCL

When set jointly with **OM_CREATE**, create a new message queue immediately without attempting to open an existing message queue. An error condition is returned if a message queue 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 message queue is created. If set, the message queue will be deleted during the last **msgQClose( )** call, independently on 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.

RETURNS
The **MSG_Q_ID** of the opened message queue, or **ERROR** if unsuccessful.

ERRNO
S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP to create the message queue.

S_msgQLib_ILLEGAL_OPTIONS
An option bit other than the options described in **msgQCreate( )** was specified.

S_msgQLib_INVALID_MSG_LENGTH
Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT
Negative maxMsgs specified.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the message queue handle.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument. *name* buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The operation attempted to create an unamed public message queue.

S_objLib_OBJ_NAME_CLASH
Both the **OM_CREATE** and **OM_EXCL** flags were set in the *mode* argument and a message queue with *name* already exists.

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.

ENOSYS
The component **INCLUDE_MSG_Q** has not been configured into the kernel

SEE ALSO
msgQLib, **msgQCreate( )**, **msgQClose( )**, **msgQUnlink( )**

_rtpSigqueue( )

NAME
_rtpSigqueue( ) – send a queued signal to an RTP with a specific signal code (syscall)

SYNOPSIS
```
int rtpSigqueue
    (
    int rtpId,
    int signo,
```

```
    const union sigval value,
    int   sigCode
    )
```

DESCRIPTION

This routine sends the signal *signo* with the signal-parameter value *value* to the process *rtpId*. The signal is sent with the signal code *sigCode*. Any task in the target RTP that has unblocked *signo* can receive the signal. 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 RTP ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO

EINVAL
EAGAIN

SEE ALSO

sigLib, **_sigqueue( )**

_sdCreate( )

NAME

_sdCreate( ) – Create a new shared data region (system call)

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_t     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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute
SD_ATTR_RX	Read/Execute
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.

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_CACHE_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 **ERROR** on failure.

ERRNOS 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_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

ENOSYS - **INCLUDE_SHARED_DATA** has not been configured into the kernel.

SEE ALSO **sdLib**, **sdOpen( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

_sdOpen( )

NAME **_sdOpen( )** – Open a shared data region for use (system call)

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_t      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

Parameter	Required?	Default
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, excepting *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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute

Attribute	Meaning
SD_ATTR_RX	Read/Execute
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.

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_CACHE_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 **ERROR** on failure.

ERRNOS

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_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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO **sdLib**, **sdCreate( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

_semGive( )

NAME **_semGive( )** – give a kernel semaphore (system call)

SYNOPSIS

```
STATUS _semGive
    (
    SEM_ID semId   /* kernel semaphore id */
    )
```

DESCRIPTION This system call performs the give operation on the specified kernel 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.

WARNING The semaphore id which is used must be the id returned from the **_semOpen( )** system call. The semaphore ids in the kernel space are distinct from the values returned by **_semOpen( )** and cannot be used with this system call.

RETURNS **OK** on success or **ERROR** otherwise

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_semLib_INVALID_OPERATION
Current task not owner of mutex 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 **semLib**, **semBLib**, **semCLib**, **semMLib**, **semEvStart( )**, **_semOpen( )**, the VxWorks programmer guides.

_semOpen( )

NAME

_semOpen( ) – open a kernel semaphore (system call)

SYNOPSIS

```
SEM_ID _semOpen
   (
   const char * name,           /* kernel semaphore name */
   SEM_TYPE     type,           /* type of semaphore */
   int          initState,      /* initial state or initial count */
   int          options,        /* semaphore options */
   int          mode            /* object management mode bits */
   void *       context         /* context value */
   )
```

DESCRIPTION

This system call either opens an existing kernel semaphore or creates a new kernel semaphore if the appropriate flags in the *mode* parameter are set. A kernel semaphore with the name *name* is searched for and if found the **SEM_ID** of the kernel semaphore is returned. A new semaphore may only be created if the search of existing kernel 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.

The parameters to the _semOpen system call are as follows:

name

an optional text string which represents the name by which the semaphore is known by. **NULL** may be specified if no name is to 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 system 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( )** (**objDelete( )**) call, independently on whether **semUnlink( )** (**objUnlink( )**) 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.

WARNING

Semaphores created by directly invoking the **_semOpen( )** system call, rather than by calling the library function, **semOpen( )** do *not* result in a user-level semaphore structure being assigned. Thus, the following list of **semLib** APIs cannot be called from a semaphore created by a direct call to **_semOpen( )**: **semTake( )**, **semGive( )**, **semDelete( )**, **semTerminate( )**, **semDestroy( )**, **semFlush( )**, **semClose( )**, **semUnlink( )**.

RETURNS

The **SEM_ID** of the opened semaphore, or **ERROR** if unsuccessful.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP to open the semaphore.

S_semLib_INVALID_OPTION
Invalid option was passed for semaphore creation.

S_semLib_INVALID_STATE
Invalid initial state for binary semaphore creation.

S_semLib_INVALID_INITIAL_COUNT
The specified initial count for counting semaphore is negative.

S_semLib_INVALID_QUEUE_TYPE
Invalid type of semaphore queue specified.

S_semLib_INVALID_OPERATION
Invalid type of semaphore requested.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the semaphore handle.

S_objLib_OBJ_INVALID_ARGUMENT
name buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

SEE ALSO **semLib**, **_semTake( )**, **_semGive( )**, **semCtl( )**, the VxWorks programmer guides.

_semTake( )

NAME **_semTake( )** – take a kernel semaphore (system call)

SYNOPSIS

```
STATUS _semTake
   (
   SEM_ID semId,     /* kernel semaphore id */
   int    timeout    /* semaphore timeout value */
   )
```

DESCRIPTION This system call performs the take operation on the specified kernel 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.

WARNING The semaphore id which is used must be the id returned from the **_semOpen( )** system call. The semaphore ids in the kernel space are distinct from the values returned by **_semOpen( )** and cannot be used with this system call.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_UNAVAILABLE,
Would have blocked but **NO_WAIT** was specified.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_semLib_INVALID_OPTION
Semaphore type is invalid

EINTR
Signal received while blocking on the semaphore

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **_semOpen( )**, the VxWorks programmer guides.

_sigqueue( )

NAME **_sigqueue( )** – send a queued signal to a RTP with a specific signal code (syscall)

SYNOPSIS

```
int _rtpSigqueue
  (
  RTP_ID                 rtpId,
  int                    signo,
  const union sigval *  pValue,
  int                    sigCode
  )
```

DESCRIPTION The routine **_sigqueue( )** sends the signal *signo* with the signal-parameter value *pValue* to the process *rtpId*. The signal sent has the signal code set to *sigCode*. Any task in the target RTP that has unblocked *signo* can receive the signal.

RETURNS **OK** (0), or **ERROR** (-1) if the RTP ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**
EAGAIN

SEE ALSO **sigLib**, **_rtpSigqueue( )**

_taskOpen( )

NAME **_taskOpen( )** – open a task (system call)

SYNOPSIS

```
int _taskOpen
    (
    VX_TASK_OPEN_SC_ARGS *   pArgs
    )
```

2

DESCRIPTION

The **VX_TASK_OPEN_SC_ARGS** structure is defined as follows:

```
typedef struct vx_task_open_sc_args
    {
    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;     /* location of execution stack */
    int          stackSize;      /* execution stack size (bytes) */
    BOOL *       pTaskCreated;   /* new kernel task created? */
    void *       context;        /* context value */
    FUNCPTR      entryPt;        /* entry point of new task */
    int          argc;           /* number of arguments to entry point */
    char **      argv;           /* arguments to application entry point */
    } VX_TASK_OPEN_SC_ARGS;
```

This system call opens a task. It searches the task name space for the first matching task. If a matching task is found, the routine returns an object handle. If a matching task is not found but the **OM_CREATE** flag is specified in the **mode** parameter, a task is created.

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.

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

Arguments to the **_taskOpen( )** system call are passed using the **VX_TASK_OPEN_SC_ARGS** structure. The following is a description of the various fields of the structure:

name

A task may be given a name as a debugging aid. This name appears in kernel shell facilities such as **i( )**. The name may be of arbitrary length and content. If the task name is specified as **NULL**, an ASCII name is given to the task of the form **t***n* where *n* is a number which increments as tasks are spawned. 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_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 user stack area. 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 user stack area. 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.

pTaskCreated
A pointer to a BOOLEAN variable used by the kernel to indicate whether a task was actually created as a result of the system call.

context
The context value assigned to the created task. This value is not actually used by VxWorks. Instead, the context value is available for OS extensions to implement facilities such as object permissions.

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 invoked with **argc** and **argv** as the parameters. Should the specified **main** routine return, a call to the kernel **exit( )** routine 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.

WARNING

Tasks created by directly invoking the **_taskOpen( )** system call, rather than by calling one of the library functions, **taskOpen( )**, **taskSpawn( )**, or **taskCreate( )**, do **not** result in a user-level task control block being assigned. Thus, the following list of **taskLib** APIs cannot be called from a task created by a direct call to **_taskOpen( )**: **taskExit( )**, **taskRtpLock( )**, **taskRtpUnlock( )**, **taskSafe( )**, **taskUnsafe( )**, and **taskIdSelf( )**. Also, the *tid* parameter to the following list of **taskLib** APIs cannot refer to a task created by a direct call to **_taskOpen( )**: **taskDelete( )**, **taskDeleteForce( )**, **taskRestart( )**, and **taskName( )**.

In addition, the task create and delete hook functions registered using **taskCreateHookAdd( )** and **taskDeleteHookAdd( )**, respectively, will not be executed for tasks created by a direct call to the **_taskOpen( )** system call.

RETURNS

The task ID, or **ERROR** if unsuccessful.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP 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 an illegal location for the user stack.

S_taskLib_ILLEGAL_OPTIONS
The operation attempted to specify an unsupported option.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the task handle.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid. *name* buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control. *pStackBase* is provided, not **NULL**, it has the same problem as *name* buffer above; Or it does belong to this RTP task but not allow to read and write. *pTaskCreated* is **NULL**; Or it has the same problem as above; Or it does not allow to write.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The operation attempted to create an unamed public task.

S_objLib_OBJ_NOT_FOUND
The **OM_CREATE** flag was not set in the **mode** argument and a task matching **name** was not found.

SEE ALSO **taskLib**, **taskSpawn( )**, **taskCreate( )**, **taskActivate( )**

_taskSigqueue( )

NAME **_taskSigqueue( )** – send queued signal to an RTP task with specific signal code (syscall)

SYNOPSIS

```
int _taskSigqueue
  (
  int                     taskId,
  int                     signo,
  const union sigval *    pValue,
  int                     sigCode
  )
```

DESCRIPTION This routine sends the signal *signo* with the signal parameter value pointed to by *pValue* to the RTP task *taskId*. The signal sent has the signal code set to *sigCode*.

RETURNS **OK** (0), or **ERROR** (-1) if the *taskId* or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**
EAGAIN

SEE ALSO **sigLib**

_timer_open( )

NAME

_timer_open( ) – open a kernel POSIX timer (system call)

SYNOPSIS

```
OBJ_HANDLE _timer_open
   (
   const char *       name,
   int                mode,
   clockid_t          clockId,
   struct sigevent *  evp,
   void *             context
   )
```

DESCRIPTION

This routine opens a kernel 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.

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

This flag is currently ignored on timers.

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 process or 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.

WARNING

Timers created by directly invoking the **_timer_open( )** system call, rather than by calling **timer_open( )** library function will not be able to use the timer library functions to operate the timer.

RETURNS

timer ID on success. Otherwise **ERROR**.

ERRNO

EINVAL
The name is not specified or the *clockId* specified is not valid.

EAGAIN
There is not enough resources to handle the request.

ENOSYS
The component **INCLUDE_POSIX_TIMERS** has not been configured into the kernel.

SEE ALSO

timerLib

_vxAtomicOr( )

NAME

_vxAtomicOr( ) – atomically perform a bitwise OR on memory location

SYNOPSIS

```
atomicVal_t vxAtomicOr
    (
    atomic_t *  target,
    atomicVal_t value
    )
```

DESCRIPTION

This routine atomically performs a bitwise OR operation of **target* and *value*, placing the result in **target*.

RETURNS

Contents of **target* before the atomic operation

ERRNO

N/A

SEE ALSO

vxAtomicLib

access( )

NAME

access( ) – determine accessibility of a file

SYNOPSIS

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

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**

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 alarm
    (
    unsigned secs
    )
```

DESCRIPTION

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

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

NOTE

64 bit value for the *secs* argument is not supported.

RETURNS

Time remaining until a previously scheduled alarm was due to be delivered, zero if there was no previous alarm, or **ERROR** otherwise.

ERRNO

N/A

SEE ALSO

timerLib

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.

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( )**

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( )**

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**

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**

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 entries in the specified cache according to the address and number of bytes parameters.

RETURNS **OK**, or **ERROR** if the operation could not be performed.

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) entries in the specified cache according to the address and number of bytes parameters. Depending on the cache design, this operation may also invalidate the cache tags.

RETURNS **OK**, or **ERROR** if the operation could not be performed.

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 entries in the specified cache according to the address and number of bytes parameters. Depending on the cache design, the invalidation may be similar to the flush.

RETURNS **OK**, or **ERROR** if the operation could not be performed.

ERRNO

S_cacheLib_INVALID_CACHE
the cache type specified is invalid.

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 operation could not be performed.

ERRNO

Not Available

SEE ALSO

cacheLib

calloc( )

NAME

calloc( ) – allocate space for an array from the RTP heap (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:

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

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

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*.

cfree( )

NAME

cfree( ) – free a block of memory from the RTP 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

memLib

chdir( )

NAME

chdir( ) – change working directory (syscall)

SYNOPSIS

```
STATUS chdir
    (
    const char * name
    )
```

DESCRIPTION

This routine sets the default I/O path. All relative pathnames specified to the I/O system will be prepended with this new current working directory, CWD. *name* can be absolute or relative to the present CWD.

RETURNS

OK, or **ERROR** if it fails to set new working directory.

ERRNO

Not Available

SEE ALSO

ioLib, **ioDefPathGet( )**, **ioDefPathSet( )**, **getpwd( )**

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 name of the file to change mode to */
    mode_t        mode    /* permission bits to assign */
    )
```

DESCRIPTION
The chmod utility changes or assigns the mode of a file. The mode of a file specifies its permissions and other attributes.

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.

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 by *res*. If *res* is **NULL**, the clock resolution is not returned.

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

ERRNO **EINVAL**
The clock ID is invalid.

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 unsuccessful.

ERRNO **EINVAL**
The specified *clock_id* is invalid.

EFAULT
The specified *tp* argument is invalid.

SEE ALSO **clockLib**

clock_nanosleep( )

NAME **clock_nanosleep( )** – high resolution sleep with specifiable clock

SYNOPSIS

```
int clock_nanosleep
    (
    clockid_t               clock_id,  /* 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 always returns **OK**.

NOTE
Non-POSIX.

RETURNS
OK always.

ERRNO
N/A

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 unsuccessful.

ERRNO **EINVAL**

The *clock_id* is invalid, *tp* is outside the supported range, the *tp* nanosecond value is less than 0 or equal to or greater than 1,000,000,000, or *clock_id* was set to **CLOCK_MONOTONIC**.

EPERM

Caller does not have the appropriate privilege to set the specified clock. User side code does not have privilege to change the realtime clock, that is, when *clock_id* is set to **CLOCK_REALTIME**.

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

---

close( )

NAME **close( )** – close a file

SYNOPSIS

```
int close
    (
    int fd
    )
```

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( )**

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**

confstr( )

NAME

confstr( ) – get strings associated with system variables

SYNOPSIS

```
size_t confstr
    (
    int    name,   /* system variable name */
    char * buf,    /* where to store the string content */
    size_t len     /* size of the buffer */
    )
```

DESCRIPTION

This routine allows an application to determine the current value of a system variable for which the value is a string.

The string content is copied into the *buf* buffer up to *len* bytes, including the terminating null character. If *len* is smaller that the actual length of the string the string is truncated to *len* - 1 bytes and is null-terminated. When this happens the value returned by **confstr( )** is greater than *len*.

If *len* is zero or *buf* is the **NULL** pointer the string content is not provided to the caller but the size of the required buffer is returned by **confstr( )**.

The supported system variables and corresponding names are listed in the table below:

System variable	Name Argument	Comments
PATH	_CS_PATH	System's default path
-	_CS_POSIX_V6_ILP32_OFF32_CFLAGS	No value
-	_CS_POSIX_V6_ILP32_OFF32_LDFLAGS	No value
-	_CS_POSIX_V6_ILP32_OFF32_LIBS	No value
-	_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS	Empty string
-	_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS	Empty string
-	_CS_POSIX_V6_ILP32_OFFBIG_LIBS	Empty string
-	_CS_POSIX_V6_LP64_OFF64_CFLAGS	No value
-	_CS_POSIX_V6_LP64_OFF64_LDFLAGS	No value
-	_CS_POSIX_V6_LP64_OFF64_LIBS	No value
-	_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS	No value
-	_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS	No value
-	_CS_POSIX_V6_LPBIG_OFFBIG_LIBS	No value
-	_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS	_POSIX_V6_ILP32_OFFBIG
-	_CS_XBS5_ILP32_OFF32_CFLAGS	No value
-	_CS_XBS5_ILP32_OFF32_LDFLAGS	No value

System variable	Name Argument	Comments
-	_CS_XBS5_ILP32_OFF32_LIBS	No value
-	_CS_XBS5_ILP32_OFF32_LINTFLAGS	No value
-	_CS_XBS5_ILP32_OFFBIG_CFLAGS	No value
-	_CS_XBS5_ILP32_OFFBIG_LDFLAGS	No value
-	_CS_XBS5_ILP32_OFFBIG_LIBS	No value
-	_CS_XBS5_ILP32_OFFBIG_LINTFLAGS	No value
-	_CS_XBS5_LP64_OFF64_CFLAGS	No value
-	_CS_XBS5_LP64_OFF64_LDFLAGS	No value
-	_CS_XBS5_LP64_OFF64_LIBS	No value
-	_CS_XBS5_LP64_OFF64_LINTFLAGS	No value
-	_CS_XBS5_LPBIG_OFFBIG_CFLAGS	No value
-	_CS_XBS5_LPBIG_OFFBIG_LDFLAGS	No value
-	_CS_XBS5_LPBIG_OFFBIG_LIBS	No value
-	_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS	No value

RETURNS

The size of the string including the terminating null character, or zero when the *name* parameter is invalid or does not have a configuration-defined value. In the latter case the errno is not changed.

ERRNO

EINVAL
when the value of the *name* argument is not valid.

SEE ALSO

confstr, **sysconf( )**

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.

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.

creat( )

NAME

creat( ) – create a file

SYNOPSIS

```
int creat
   (
   const char * name,
   mode_t mode
   )
```

DESCRIPTION

This routine creates a file called *name* and opens it with a specified *mode*. 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 *mode* is set to **O_RDONLY** (0), **O_WRONLY** (1), or **O_RDWR** (2) for the duration of time the file is open. On NFS and POSIX compliant file systems such as HRFS, *mode* 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.

ENODEV
No valid device name found in path.

others
Other errors reported by device drivers.

SEE ALSO **ioLib**, **open( )**

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**

dlclose( )

NAME **dlclose( )** – unlink the shared object from the RTP's address space

SYNOPSIS

```
int dlclose
    (
    void *  handle     /* handle of shared object to close */
    )
```

DESCRIPTION **dlclose( )** unlinks and removes the object referred to by handle from the process address space. If multiple calls to **dlopen( )** have been done on this object (or the object was one loaded at startup time) the object is removed when its reference count drops to zero.

RETURNS -1 if the handle is invalid; 0 on success

ERRNO Not Available

SEE ALSO **rtld**

dlerror( )

NAME **dlerror( )** – get most recent error on a call to a dynamic linker routine

SYNOPSIS

```
char *dlerror
    (
                            /* this routine takes no arguments */
    )
```

DESCRIPTION **dlerror( )** returns a character string representing the most recent error that has occurred while processing one of the other functions described here. If no dynamic linking errors have occurred since the last invocation of **dlerror( )**, **dlerror( )** returns **NULL**. Thus, invoking **dlerror( )** a second time, immediately following a prior invocation, will result in **NULL** being returned.

RETURNS A string, possibly **NULL**

ERRNO Not Available

SEE ALSO **rtld**

dlopen( )

NAME **dlopen( )** – map the named shared object into the RTP's address space

SYNOPSIS

```
void * dlopen
    (
    const char *  name,
    int           mode
    )
```

DESCRIPTION The **dlopen( )** function takes a name of a shared object as the first argument. The shared object is mapped into the address space, relocated and its external references are resolved in the same way as is done with the implicitly loaded shared libraries at program startup.

PARAMETERS

name
The name of the shared object The name can either be an absolute pathname or it can be of the form **`lib***name*.so[.xx[.yy]]' in which case the same library search rules apply that are used for shared library searches. If the first argument is **NULL**, **dlopen( )** returns a handle on the global symbol object. This object provides access to all symbols from an ordered set of objects consisting of the original program image and any dependencies loaded during startup.

mode
This must be either **RTLD_LAZY**. meaning symbols are resolved as and when code from the shared object is executed, or **RTLD_NOW** meaning all undefined symbols are resolved before dlopen returns and the call fails if this cannot be done **RTLD_GLOBAL** may optionally be or'ed with mode, in which case the external symbols defined in the shared object will be made available to subsequently loaded shared objects

dlopen( ) returns a handle to be used in calls to **dlclose( )**, and **dlsym( )**. If the named shared object has already been loaded by a previous call to **dlopen( )** (and not yet unloaded by **dlclose( )**), a handle referring to the resident copy is returned.

RETURNS Handle to the loaded object, or **NULL** on failure

ERRNO Not Available

SEE ALSO **rtld**

dlsym( )

NAME **dlsym( )** – resolve the symbol defined in the shared object to its address

SYNOPSIS

```
void * dlsym
    (
    void *        handle,    /* handle of shared object */
    const char *  name       /* name of symbol to resolve */
    )
```

DESCRIPTION **dlsym( )** looks for a definition of symbol in the shared object designated by handle. The symbols address is returned. If the symbol cannot be resolved, **NULL** is returned.

RETURNS The symbol's address, or **NULL** if it cannot be resolved

ERRNO Not Available

SEE ALSO **rtld**

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.

dup( )

NAME
dup( ) – duplicate a file descriptor (syscall)

SYNOPSIS

```
int dup
    (
    int fd
    )
```

DESCRIPTION
Duplicate an open file descriptor. This command is used to duplicate a file descriptor entry with a new file descriptor number. Upon completion, any reference to the new file descriptor number is the same as a reference to the original file descriptor number. The current file pointer is shared by the two open file descriptors. Read/Write activity on either one will advance the current pointer.

The returned file descriptor number is the first available number from the file descriptor table. If the table is full, then **ERROR** is returned with errno set to **EMFILE**.

RETURNS
New file descriptor number, or **ERROR** if the input number is not an open file descriptor.

ERRNO

EBADF
The *fd* argument is not a valid file descriptor number.

EMFILE
Maximum number of open files has been reached.

SEE ALSO
ioLib

dup2( )

NAME
dup2( ) – duplicate a file descriptor as a specified *fd* number (syscall)

SYNOPSIS

```
int dup2
    (
    int fd,
    int fd2
    )
```

DESCRIPTION Modified version of the **dup( )** command that takes a second argument which is to be the duplicated file descriptor number. If the second file descriptor number is already open, it will be closed before being reopened as a duplicate of the first *fd* number.

RETURNS Returns the duplicate file descriptor number, or **ERROR** if either argument is invalid, or if the input *fd* number is not open.

ERRNO

EBADF
The *fd* is not a valid open file descriptor.

EINVAL
The fd2 argument is not a valid number for an fd.

SEE ALSO **ioLib**

---

edrErrorInject( )

NAME **edrErrorInject( )** – injects an error into the ED&R subsystem

SYNOPSIS

```
STATUS edrErrorInject
    (
    int           kind,         /* severity | facility          */
    const char * fileName,      /* name of source file          */
    int           lineNumber,   /* line number of source code   */
    REG_SET *     pRegs,        /* pointer to REG_SET           */
    void *        address,      /* faulting address             */
    const char * msg            /* additional text string       */
    )
```

NOTE Although users are free to call the **edrErrorInject( )** function directly, a more convienient set of macros are provided in **edrLib.h**. It is recommended to use these macros (eg. **EDR_USER_FATAL_INJECT**) whenever possible.

This function passes the supplied arguments to the ED&R subsystem using the **_edrErrorInject( )** system call. The system call will store the provided information in an error record, along with other useful information, such as:

- the OS version
- the CPU type (and number, for future MP systems)
- the time at which the error occured
- the current OS context (task / interrupt / exception, RTP)
- a code fragment from around the faulting instruction
- a stack trace of the most recent stack frames

Only the RTP and USER facility levels are available from user level. The use of any other facility will return **ERROR**.

RETURNS **OK** if the error was stored correctly, or **ERROR** if some failure occurs during storage or an invalid facility is specified.

ERRORS

S_edrLib_NOT_INITIALIZED
The ED&R library was not initialized

S_edrLib_PROTECTION_FAILURE
The ED&R memory log could not be protected or unprotected

S_edrLib_INVALID_OPTION
An invalid facility or severity was provided

SEE ALSO **edrLib**

edrFlagsGet( )

NAME **edrFlagsGet( )** – return the current ED&R flags set in the kernel (system call)

SYNOPSIS

```
int edrFlagsGet (void)
```

DESCRIPTION This syscall returns all the ED&R flags which have been set to "on".

RETURNS an integer with the appropriate bits set, or **ERROR**

ERRNO Not Available

SEE ALSO **edrLib**

edrIsDebugMode( )

NAME **edrIsDebugMode( )** – determines if the ED&R debug flag is set

SYNOPSIS

```
BOOL edrIsDebugMode(void)
```

DESCRIPTION This function takes no parameters and returns a boolean indicating whether or not the ED&R debug mode flag has been set. If the flags can't be retrieved, the default state is set to off.

RETURNS **TRUE** or **FALSE** depending on the state of the debug flag

ERRNO Not Available

SEE ALSO **edrLib**

errnoGet( )

NAME **errnoGet( )** – get the error status value of the calling task

SYNOPSIS

```
int errnoGet (void)
```

DESCRIPTION This routine gets the error status value of the calling task. It is provided for compatibility with previous versions of VxWorks.`

For tasks that were created by a **taskLib** library function (**taskOpen( )**, **taskSpawn( )**, or **taskCreate( )**), **errnoGet( )** accesses the **errno** value maintained for each context separately in the task TCB.

Using **errnoGet( )** to find the error status value of tasks that were created by a direct invocation of the **_taskOpen( )** system call, rather than by calling one of the **taskLib** library functions, results in obtaining the value of the global error status variable **errno**.

RETURNS The error status value contained in **errno**, either in the task TCB or in the global variable.

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 for a specified task. If *taskId* is zero, the calling task is assumed. The value currently in the specified task TCB is returned, except for tasks created with **_taskOpen( )** where this value is not available in user space.

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 or the task was created using a direct call to the **_taskOpen( )** system call.

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 of a specified task with the specified error status. If *taskId* is zero, the calling task is assumed.

Tasks that were created by a **taskLib** library function (**taskOpen( )**, **taskSpawn( )**, or **taskCreate( )**) access the **errno** value maintained for each context separately in the task TCB.

You cannot use **errnoOfTaskSet( )** to set the error status of tasks that were created by a direct invocation of the **_taskOpen( )** system call, rather than by calling one of the **taskLib** library functions.

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 or the task was created using a direct call to the **_taskOpen( )** system call.

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 error status value of the calling task with a specified error status. It is provided for compatibility with previous versions of VxWorks.

Tasks that were created by a **taskLib** library function (**taskOpen( )**, **taskSpawn( )**, or **taskCreate( )**) access the **errno** value maintained for each context separately in the task TCB.

Using **errnoSet( )** to set the error status of tasks that were created by a direct invocation of the **_taskOpen( )** system call, rather than by calling one of the **taskLib** library functions, results in updating the value of the global error status variable **errno**.

RETURNS **OK**

ERRNO N/A

SEE ALSO **errnoLib**, **errnoGet( )**, **errnoOfTaskSet( )**

eventClear( )

NAME **eventClear( )** – Clear all events for calling task

SYNOPSIS

```
STATUS eventClear
    (
    void
    )
```

DESCRIPTION This function clears all received events for the calling task.

RETURNS **OK** on success or **ERROR**.

ERRNO **S_intLib_NOT_ISR_CALLABLE**
Routine has been called from interrupt level.

SEE ALSO **eventLib**

eventReceive( )

NAME

eventReceive( ) – Receive event(s) for the calling task

SYNOPSIS

```
STATUS eventReceive
    (
    UINT32   events,           /* events task is waiting to occur */
    UINT8    options,          /* user options */
    int      timeout,          /* ticks to wait */
    UINT32 * pEventsReceived /* events occured are returned through this */
    )
```

DESCRIPTION

This function is called to receive event(s) for the calling task. It may pend task until one or all specified *events* have occurred based on option and timeout parameters.

The parameter *pEventsReceived* is always filled with the events received completely or partially even when the function returns an error, provided an valid address is passed. This is the best effort event receiving.

The *options* parameter is used for four user options. Firstly, it is used to specify if the task is going to wait for all events to occur or only one of them. One of the following has to be selected:

EVENTS_WAIT_ANY (0x1)
: only one event has to occur

EVENTS_WAIT_ALL (0x0)
: will wait until all events occur.

Secondly, it is used to specify if the events returned in *pEventsReceived* will be only those received and wanted, or all events received (even the ones received before **eventReceive( )** was called). By default it returns only the events wanted.

EVENTS_RETURN_ALL (0x2)
: When this option is turned on, it causes the function to return received events, both wanted and unwanted. All events are cleared when this option is selected.

Thirdly, the user can specify if the events received but not wanted are to be cleared or not in the calling task's events register. They are cleared by default. Wanted events are always cleared.

EVENTS_KEEP_UNWANTED (0x4)
: Tells the system not to clear the unwanted events. In the case that the option **EVENTS_RETURN_ALL** is used, all events are cleared even if this one is selected.

Lastly, it can be used to retrieve what events have been received by the current task.

EVENTS_FETCH (0x80)
: If this option is set, then *pEventsReceived* will be filled with the events that have already been received and will return immediately. In this case, the parameters *events* and

timeout, as well as all the other options, are ignored. Also, events are not cleared, allowing to get a peek at the events that have already been received.

The *timeout* parameter specifies the number of ticks to wait for wanted events to be sent to the waiting task. 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.

WARNING
This routine may not be used from interrupt level.

RETURNS
OK on success or **ERROR**.

ERRNO
S_eventLib_TIMEOUT
Wanted events not received before specified time expired.

S_eventLib_NOT_ALL_EVENTS
Specified **NO_WAIT** as the timeout parameter and wanted events were not already received when the routine was called.

S_eventLib_ZERO_EVENTS
The *events* parameter has been passed a value of 0.

S_objLib_OBJ_DELETED
Task is waiting for some events from a resource that is subsequently deleted.

S_objLib_OBJ_INVALID_ARGUMENT
pEventsReceived is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be written due to access control.

S_intLib_NOT_ISR_CALLABLE
Function has been called from ISR.

SEE ALSO
eventLib, **semEvLib**, **msgQEvLib**, **eventSend( )**

eventSend( )

NAME
eventSend( ) – Send event(s) to a task

SYNOPSIS
```
STATUS eventSend
```

```
    (
    int    taskId,    /* task events will be sent to */
    UINT32 events     /* events to send               */
    )
```

DESCRIPTION This is called to send specified event(s) to specified task. Passing a taskId of **NULL** sends events to the calling task. This function never blocks.

RETURNS **OK** on success or **ERROR**.

ERRNO **S_objLib_OBJ_ID_ERROR**
Task ID is invalid.

S_eventLib_NULL_TASKID_AT_INT_LEVEL
Routine was called from ISR with a taskId of **NULL**.

SEE ALSO **eventLib**, **eventReceive( )**

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**

2

fchmod( )

NAME

fchmod( ) – change the permission mode of a file

SYNOPSIS

```
int fchmod
    (
     int fd,                /* file descriptor for changed mode */
     mode_t mode            /* permission bits to assign */
    )
```

DESCRIPTION

The fchmod function changes or assigns the mode of a file. The mode of a file specifies its permissions and other attributes.

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.

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
fsPxLib

fdatasync( )

NAME

fdatasync( ) – synchronize a file data

SYNOPSIS

```
int fdatasync
   (
    int fd      /* file descriptor of the file to sync for data integrity  */
   )
```

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( )**.

RETURNS

The number of characters output, or **ERROR** if there is an error during output.

ERRNO

Not Available

SEE ALSO

fioLib, **printf( )**

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**

2

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

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.

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*.

RETURNS

The number of bytes read, or **ERROR** if there is an error during the read operation.

ERRNO

Not Available

SEE ALSO

fioLib, **read( )**

fpathconf( )

NAME

fpathconf( ) – determine the current value of a configurable limit

SYNOPSIS

```
long fpathconf
    (
     int fd,     /* file descriptor of the file to query  */
     int name    /* name represents the variable to be queried  */
    )
```

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( )**

---

free( )

NAME **free( )** – free a block of memory from the RTP heap (ANSI)

SYNOPSIS

```
void free
     (
     void * ptr  /* pointer to block of memory to free */
     )
```

DESCRIPTION This routine returns to the free memory pool of the RTP heap a block of memory previously allocated with **malloc( )**, valloc, **memalign( )** or **calloc( )**.

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 **memLib**, **malloc( )**, **calloc( )**, **memPartFree( )**, *American National Standard for Information Systems -, Programming Language - C, ANSI X3.159-1989: General Utilities* (**stdlib.h**)

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.

If fildes refers to a Shared Memory Object, **ftruncate( )** shall set the size of the shared memory object to length.
If the effect of **ftruncate( )** is to decrease the size of a Shared Memory Object or Memory Mapped File and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded. References to discarded pages would be possible but, msync on the discarded pages will not succeed.

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. Length cannot be completely represented with an **off_t** type.

SEE ALSO

ftruncate

getOptServ( )

NAME

getOptServ( ) – parse parameter string into argc, argv format

SYNOPSIS

```
int getOptServ
    (
    char * ParamString,
    const char * progName,      /* program name value for argv[0] */
    int * argc,
```

```
    char * argvloc[],
    int argvlen
    )
```

DESCRIPTION none

RETURNS 0 (**OK**) if all arguments were successfully stored; otherwise, -1 (**ERROR**).

ERRNO Not Available

SEE ALSO **getopt**

---

getcwd( )

NAME **getcwd( )** – get pathname of current working directory

SYNOPSIS

```
char * getcwd
    (
    char * buffer,
    size_t length
    )
```

DESCRIPTION The **getcwd( )** function copies the current working directory pathname into a user supplied buffer. The value of *length* must be at least one greater than the length of the pathname to be returned.

Currently, *buffer* must be non-**NULL**. In the future, passing a **NULL** pointer will cause **getcwd( )** to malloc a buffer in which to store the path.

RETURNS pointer to the user supplied buffer, or **NULL** if the buffer is invalid, or too small for the pathname.

ERRNO

EINVAL
invalid arguments.

ERANGE
Buffer is not large enough to receive the path name.

SEE ALSO **ioLib**, **ioDefPathGet( )**, **ioDefPathSet( )**, **chdir( )**

getenv( )

NAME

getenv( ) – get value of an environment variable (POSIX)

SYNOPSIS

```
char * getenv
    (
    const char * envVarName  /* name of environment variable to find */
    )
```

DESCRIPTION

This routine searches the environment of the RTP for the environment variable *envVarName* and returns its value if the variable exists.

Note that the string pointed to by the **getenv( )** function can be modified by a subsequent call to **setenv( )** or **unsetenv( )**.

RETURNS

a pointer to the value of the environment variable, or **NULL** if the variable does not exist.

ERRNO

N/A

SEE ALSO

getenv, **setenv( )**, **unsetenv( )**

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

getpid( )

NAME **getpid( )** – Get the process identifier for the calling process (syscall)

SYNOPSIS

```
pid_t getpid
   (
   void
   )
```

DESCRIPTION This routine gets the process identifier for the calling process. The ID is guaranteed to be unique and is useful for constructing uniquely named entities such as temporary files etc.

RETURNS Process identifier for the calling process.

ERRNO N/A.

SEE ALSO **rtpLib**, the VxWorks programmer guides

getppid( )

NAME **getppid( )** – Get the parent process identifier for the calling process (syscall)

SYNOPSIS

```
pid_t getppid
   (
   void
   )
```

DESCRIPTION This routine gets the process identifier for the parent of the calling process. The ID is guaranteed to be unique and is useful for constructing uniquely named entities such as temporary files etc.

If the parent of the calling task's RTP has terminated or that the calling task's parent is the kernel, then **NULL** will be returned to indicate that the parent of the process is the kernel.

RETURNS Process identifier for the parent of the calling process, or **NULL**

ERRNO N/A.

SEE ALSO **rtpLib**, the VxWorks programmer guides

getprlimit( )

NAME **getprlimit( )** – get process resource limits (syscall)

SYNOPSIS

```
int getprlimit
    (
    int              idtype,
    RTP_ID           id,
    int              resource,
    struct rlimit *  rlp
    )
```

DESCRIPTION none

RETURNS 0 on success, -1 on errors.

ERRNO

EFAULT
The address specified for rlp is invalid.

EINVAL
invalid arguments.

SEE ALSO **ioLib**

getwd( )

NAME **getwd( )** – get the current default path

SYNOPSIS

```
char*  getwd
    (
    char* pathname  /* where to return the pathname */
    )
```

DESCRIPTION This routine copies the name of the current default path to *pathname*. It provides the same functionality as **ioDefPathGet( )** and **getcwd( )**. It is provided for compatibility with some older UNIX systems.

The parameter *pathname* should be **MAX_FILENAME_LENGTH** characters long.

RETURNS A pointer to the resulting path name.

ERRNO
Not Available

SEE ALSO
ioLib

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

2

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( )**

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

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.

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( )**.

---

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**

---

ioDefPathGet( )

NAME **ioDefPathGet( )** – get the current default path (VxWorks)

SYNOPSIS

```
void ioDefPathGet
    (
    char *buffer  /* where to return the pathname */
    )
```

DESCRIPTION This routine copies the name of the current default path to *buffer*. It provides the same functionality as **getcwd( )** and is provided for backward compatibility.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **ioLib**, **chdir( )**, **getcwd( )**, **ioDefPathSet( )**

---

ioDefPathSet( )

NAME **ioDefPathSet( )** – vxWorks compatible ioDefPathSet (chdir)

SYNOPSIS

```
STATUS ioDefPathSet
    (
    const char *pathname  /* name of the new default path */
    )
```

DESCRIPTION This routine sets the default I/O path. All relative pathnames specified to the I/O system will be prepended with this pathname. This pathname must be an absolute pathname, i.e., *name* must begin with an existing device name.

RETURNS **OK**, or **ERROR** if the first component of the pathname is not an existing device.

ERRNO Not Available

SEE ALSO **ioLib**, **chdir( )**, **getcwd( )**, **ioDefPathGet( )**

---

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.

ioctl( )

NAME **ioctl( )** – perform an I/O control function

SYNOPSIS

```
int ioctl
    (
    int fd,
    int function,
    ...
    )
```

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.

RETURNS The return value of the driver, or **ERROR** if the file descriptor does not exist.

ERRNO

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**, **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rawFsLib**, the VxWorks programmer guides

isatty( )

NAME

isatty( ) – return whether the underlying driver is a *tty* device

SYNOPSIS

```
int isatty
    (
    int fd  /* file descriptor to check */
    )
```

DESCRIPTION

This routine simply invokes the **ioctl( )** function FIOISATTY on the specified file descriptor.

RETURNS

1 (**TRUE**), or 0 (**FALSE**) if the driver does not indicate a *tty* device.

ERRNO

Not Available

SEE ALSO

ioLib

kill( )

NAME

kill( ) – send a signal to an RTP (syscall)

SYNOPSIS

```
int kill
  (
  pid_t rtpId,
  int signo
  )
```

DESCRIPTION

This routine sends a signal *signo* to the RTP specified by *rtpId*. Any task in the target RTP that has unblocked *signo* can receive the signal. This API can also be used to send signals to public tasks in other RTP's. If *rtpId* is **NULL** the signal will be senting to the caller's RTP.

This is a POSIX specified routine.

RETURNS

OK (0), or **ERROR** (-1) if the RTP ID or signal number is invalid.

ERRNO

EINVAL
The value of *sig* is an invalid or unsupported signal number.

ESRCH
The RTP *rtpId* can not be found.

SEE ALSO

sigLib, **rtpKill( )**, **taskKill( )**

link( )

NAME **link( )** – link a file

SYNOPSIS

```
int link
    (
    const char *name,   /* path name of the file to be linked */
    const char *newname /* path name with which to link to */
    )
```

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( )**.

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.

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 kernel 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*

SEE ALSO **loginLib**

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**.

ERRORS **EINVAL** is set if the whence parameter is invalid, or if offset is larger than a 32-bit number. Other errors may be set by the io system or device drivers.

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

malloc( )

NAME
malloc( ) – allocate a block of memory from the RTP heap (ANSI)

SYNOPSIS
```
void * malloc
    (
    size_t nBytes  /* number of bytes to allocate */
    )
```

DESCRIPTION
This routine allocates a block of memory from the free list of the RTP 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:

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

SEE ALSO
memLib, **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 RTP 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 RTP memory partition, after the initial allocation of memory to the RTP memory partition.

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

memLib, **memPartAddToPool( )**

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( )**

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**

memFindMax( )

NAME **memFindMax( )** – find the largest free block in the RTP heap

SYNOPSIS

```
int memFindMax (void)
```

DESCRIPTION This routine searches for the largest block in the current heap free list and returns its size. It returns 0 if there is no free block in the memory partition.

If the RTP heap's autogrowth is enabled, it is possible to allocate a block larger than the value returned by this routine.

RETURNS The size, in bytes, of the largest available block.

ERRNO Not Available

SEE ALSO **memLib**, **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 **memLib**, **memPartInfoGet( )**

memOptionsGet( )

NAME **memOptionsGet( )** – get the options for the RTP heap

SYNOPSIS

```
STATUS memOptionsGet
    (
    UINT * pOptions  /* pointer to options for current heap */
    )
```

DESCRIPTION This routine sets location pointed by the parameter *pOptions* with the options of the RTP heap.

Heap/memory partition options are discussed in details in the reference entry for **memPartLib**.

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

ERRNO Not Available

SEE ALSO **memLib**, **memOptionsSet( )**, **memPartOptionsGet( )**, **memPartOptionsSet( )**

memOptionsSet( )

NAME **memOptionsSet( )** – set the options for the RTP heap

SYNOPSIS

```
STATUS memOptionsSet
    (
    unsigned options  /* options for current heap */
    )
```

DESCRIPTION This routine sets the debug and error handling options for the RTP heap. For detailed description of these options see the **memPartLib** and **memPartOptionsSet( )** references.

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_memLib_INVALID_ADDRESS
pPool is equal to **NULL**.

S_memLib_INVALID_NBYTES
poolSize value is too small.

SEE ALSO **memPartLib**, **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.

ENOMEM / 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:

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

SEE ALSO **memPartLib**, **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 is allocated out of the RTP heap partition.

RETURNS The partition ID, or **NULL** if there is insufficient memory in the RTP 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**

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 Not Available

SEE ALSO **memPartLib**, **memPartCreate( )**

memPartFindMax( )

NAME **memPartFindMax( )** – find the size of the largest 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.

If the partition's autogrowth is enabled, it is possible to allocate a block larger than the value returned by this routine.

RETURNS The size, in bytes, of the largest available block.

ERRNO Not Available

SEE ALSO **memPartLib**, **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 list 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 is invalid.

ERRNO Possible errnos generated by this routine include:

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**, **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

memPartLib, **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 Not Available

SEE ALSO **memPartLib**, **memPartOptionsSet( )**, **memOptionsGet( )**

memPartOptionsSet( )

NAME **memPartOptionsSet( )** – set the debug 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. For the list of supported options see the **memPartLib** library reference guide.

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

ERRNO Not Available

SEE ALSO **memPartLib**, **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 set to zero.

ERRNO Possible errnos generated by this routine include:

S_memLib_BLOCK_ERROR
The block of memory to free is not valid.

ENOMEM / 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 **memPartLib**, **realloc( )**

memalign( )

NAME **memalign( )** – allocate aligned memory from the RTP 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 RTP 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:

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

S_memLib_INVALID_ALIGNMENT
alignment is not a power of two.

SEE ALSO **memLib**, **memPartAlignedAlloc( )**

mkdir( )

NAME **mkdir( )** – make a directory

SYNOPSIS

```
int mkdir
    (
    const char *       dirName,        /* directory name */
    mode_t             mode            /* mode of dir */
    )
```

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. *mode* sets the initial permission bits of the new directory.

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**

mlock( )

NAME **mlock( )** – lock specified pages into memory

SYNOPSIS

```
int mlock
    (
    const void * addr,  /* address to memory block */
    size_t       len    /* size of memory block */
    )
```

DESCRIPTION This routine guarantees that the specified pages are memory resident. In VxWorks paging is not implemented, therefore all mapped pages are always memory resident. Therefore this routine only validates parameters, but has no effect on the mapped memory.

RETURNS 0, or -1 if the memory does not belong to the process.

ERRNO

ENOMEM
Some or all of the address range specified by the addr and len arguments do not correspond to valid mapped pages in the address space of the process.

SEE ALSO **mmanLib**, **munlock( )**, **mmap( )**

mlockall( )

NAME

mlockall( ) – lock all pages used by a process into memory

SYNOPSIS

```
int mlockall
    (
    int flags  /* flags for memory locking */
    )
```

DESCRIPTION

This routine guarantees that all pages used by a process are memory resident. In VxWorks memory is never paged, therefore all mapped pages are always memory resident. Therefore this routine only validates the *flags* argument, but has no effect on the mapped memory. The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants:

Flag	Meaning
MCL_CURRENT	Lock all of the pages currently mapped into the addres space of the process.
MCL_FUTURE	Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

RETURNS

0 on success, or -1 if the invalid flag parameter was passed.

ERRNO

EINVAL
The flags argument is zero, or includes unsupported flags.

SEE ALSO

mmanLib, **munlockall( )**, **mmap( )**

mmap( )

NAME

mmap( ) – map pages of memory (syscall)

SYNOPSIS

```
void * mmap
   (
   void *     addr,              /* requested address to map */
   size_t     len,               /* size of memory to be mapped */
   int        prot,              /* read/write/execute protections */
   int        flags,             /* shared/private/fixed/anon */
   int        fildes,            /* memory object file descriptor */
   off_t      off                /* offset in file */
   )
```

DESCRIPTION

This routine establishes a mapping between an RTP's address space and a regular file, shared memory object, or directly the system RAM (anonymous).

When the mapping is for a regular file or shared memory object, the mapping is performed for the object represented by the file descriptor *fildes* at offset *off* for *len* bytes.

The parameter *flags* provides information about the handling of the mapped data. The value of flags is the bitwise-inclusive OR of these options, defined in *sys/mman.h*:

Symbolic constant	Description
MAP_PRIVATE	Create a mapping that is private to the RTP.
MAP_SHARED	Create a mapping that is shared by RTPs.
MAP_ANONYMOUS	Create mapping directly to system RAM.

Notes: The **MAP_FIXED** flag is not supported. When **MAP_ANONYMOUS** is used, the *fildes* and *offset* parameters are ignored, and it must be used together with the **MAP_PRIVATE** flag.

If **MAP_PRIVATE** is specified, modifications to the mapped data by the calling process is visible only to the calling process and does not change the underlying object (file), even if **msync( )** is called. Modifications to the underlying object done after the **MAP_PRIVATE** mapping is established are not visible through the **MAP_PRIVATE** mapping, except when **msync( )** is called for memory mapped files with the **MS_INVALIDATE** option.

Either **MAP_SHARED** or **MAP_PRIVATE** can be specified, but not both.

The *prot* parameter must be either **PROT_NONE** or the bitwise-inclusive OR of one or more of the other flags in the following symbolic constants, defined in the *sys/mman.h* header:

Symbolic constant	Description
PROT_NONE	Page cannot be accessed.
PROT_READ	Page can be read.
PROT_WRITE	Page can be written.
PROT_EXEC	Page can be executed.

In VxWorks, when a page is writable or executable, it is always readable as well. On some architectures readable pages are also implicitly executable.

The *off* argument must be aligned and sized according to system's MMU page size, as returned by **sysconf( )** when passed _SC_PAGESIZE or _SC_PAGE_SIZE. **mmap( )** always performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the resulting mapping size is rounded to whole pages.

The system always zero-fills any partial page at the end of an object. Modified portions of the last page of an object which are beyond its end are never written to the object. On systems with MMU enabled, references to whole pages following the end of an object result in delivery of a SIGBUS signal.

The **mmap( )** function adds an extra reference to the file associated with the file descriptor *fildes*. This reference is not removed by a subsequent **close( )** on *fildes*. This reference is removed when there are no more mappings to the file.

The st_atime field of a mapped file are updated when **mmap( )** is called. The st_ctime and st_mtime fields of a file that is mapped with **MAP_SHARED** and **PROT_WRITE** are updated when **msync( )** is called with **MS_ASYNC** or **MS_SYNC**, or the respective file is unmapped.

When an RTP is terminated or deleted, all mapped pages are automatically unmapped.

In order to minimize context switch overhead for all processor variants, VxWorks avoids creation of aliased mappings (when the same physical memory page is mapped to multiple virtual pages). This means shared mappings of the same file or object always use the same virtual address, even when mapped in different RTPs. Because of this, an existing shared mapping of file or shared memory object cannot always be remapped with overlapping offset ranges. Subsequent **mmap( )** calls that require adjacent virtual memory pages that cannot be allocated to the process will result in **ENOMEM** error.

Although in this implementation aliasing is not allowed, applications should not rely on this behaviour. For maximum portability and forward looking compatibility, applications should always share relative references and not absolute references of mapped data.

After a file or shared memory object has been mapped with **mmap( )**, if the file is truncated, and if the effect of **ftruncate( )** is to decrease the size of a Shared Memory Object or Memory Mapped File, and if whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded. References to discarded pages would be possible but, msync on the discarded pages will not succeed.

RETURNS

The mapped address, or **MAP_FAILED** in case of error.

ERRNO

EACCES
: The *fildes* argument is not open for read, regardless of the protection specified, or fildes is not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.

EINVAL
: Invalid address, length, offset or flags parameter.

EBADF
: Invalid *fildes* parameter.

ENOTSUP
: Protection requested is not supported, or the *flags* is unsupported.

ENOMEM
: Not enough memory left in system.

ENODEV
: The *fildes* argument refers to a file whose type is not supported by **mmap( )**.

ENOMEM
: There is insufficient room in the address space to effect the mapping.

ENXIO
: Addresses in the range [*off*,*off*+*len*) are invalid for the object specified by fildes.

SEE ALSO

mmanLib, **munmap( )**, **mprotect( )**

mprobe( )

NAME **mprobe( )** – probe memory mapped in process

SYNOPSIS

```
int mprobe
    (
    void * addr,  /* address to memory block */
    size_t len,   /* size of memory block */
    int    prot   /* protection value */
    )
```

DESCRIPTION This routine verifies that memory is mapped in the address space of a process with a requested protections.

The *prot* argument is constructed from the bitwise-inclusive OR of one or more of the following flags defined in the *sys/mman.h* header:

Protection	Meaning
PROT_NONE	memory protection is ignored.
PROT_READ	memory can be read.
PROT_WRITE	memory can be written.

Note that passing **PROT_NONE** by itself only verifies that the memory is mapped in the address space of the process, disregarding protections. **PROT_READ** and **PROT_WRITE** override **PROT_NONE** when they are bitwise OR-ed together.

RETURNS 0 on success, or -1 if memory cannot be accessed.

ERRNO

ENOMEM
Some or all of the address range specified by the addr and len arguments do not correspond to valid mapped pages in the address space of the process.

EACCES
Some or all of the address range specified by the addr and len arguments cannot be accessed with the specified permission.

EINVAL
Protection parameter is not valid.

SEE ALSO **mmanLib**, **mmap( )**, **mprotect( )**

mprotect( )

NAME **mprotect( )** – set protection of memory mapping (syscall)

SYNOPSIS

```
int mprotect
    (
    void *    addr,              /* address of memory to set */
    size_t    len,               /* length of memory block to set */
    int       prot               /* new protection value */
    )
```

DESCRIPTION

This routine changes the access protections on the mappings specified by the range bounded by *addr* and *addr+len*; *len* is rounded up to the next multiple of the page size.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the mapped data. The *prot* argument should be either **PROT_NONE** or the bitwise-inclusive OR of one or more of **PROT_READ**, **PROT_WRITE**, and **PROT_EXEC**. For more information about these, see the **mmap( )** API guide.

When **mprotect( )** fails due to some pages not being mapped, a subset of the pages may get the new protection set while others don't. For example, if three pages are to be potected and the second page is not currently mapped, then the first page may get updated, and the last page may not.

RETURNS

0 on success, or -1 in case of failure.

ERRNO

EACCES
The prot argument specifies a protection that violates the access permission the process has to the underlying memory object.

EINVAL
Address is not page aligned or block size is 0.

ENOMEM
The memory block is not mapped for the RTP.

ENOTSUP
Protection requested is not supported.

SEE ALSO

mmanLib, **mmap( )**, **munmap( )**

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.

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( )**:

mq_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 message queues.

If this notification type specified in the sigev_notify field of *pNotification* is **SIGEV_THREAD** then a POSIX thread will be spawned in the calling process using the attributes specified in the sigev_notify_attributes field and the entry point specified in sigev_notify_function. The argument passed to this routine is specified in the sigev_value field. The detach state specified in sigev_notify_attributes must be **PTHREAD_CREATE_DETACHED**.

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.

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, or the *pNotification* argument is invalid.

SEE ALSO **mqPxLib**, **mq_open( )**, **mq_send( )**, **pthreadLib**

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( )**.

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 other RTPs cannot get access to it.

The *oflags* requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages shall be granted if the calling process would be granted read or write access, respectively, to an equivalently protected message queue.

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.

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

EACCES
The message queue exists and the permission specified by oflags are denied.

EEXIST
O_CREAT and **O_EXCL** are set and the message queue alread 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 messaeg queue.

EINVAL
The specified *name* is invalid, or An invalid combination of *oflags* is specified, or **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.

ENAMETOOLONG
The name of the message queue is too long.

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 */
    char     * pMsg,     /* buffer to receive message */
    size_t     msgLen,   /* size of buffer, in bytes */
    unsigned * 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. A *msgLen* size greater than **SSIZE_MAX** will also 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, **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 that has been waiting the longest 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, no message is removed from the queue, and **mq_receive( )** returns an error.

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 or greater than **SSIZE_MAX**.

EINVAL
The *pMsg* pointer is invalid.

EINTR
Signal received while blocking on the message queue.

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 char * pMsg,    /* message to send */
    size_t       msgLen,  /* size of message, in bytes */
    unsigned     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's, **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 **EAGAIN** error.

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,      /* msg queue descriptor */
    const struct mq_attr * _Restrict pMqStat,    /* new attributes */
    struct mq_attr * _Restrict 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.

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_timedreceive( )

NAME **mq_timedreceive( )** – receive a message from a message queue with timeout (POSIX)

SYNOPSIS

```
ssize_t mq_timedreceive
    (
    mqd_t                       mqdes,       /* message queue descriptor */
    char * _Restrict            pMsg,        /* buffer to receive message */
    size_t                      msgLen,      /* size of buffer, in bytes */
    unsigned * _Restrict        pMsgPrio,    /* if not NULL, priority of msg */
    const struct timespec * _Restrict abs_timeout  /* timeout to wait for */
    )
```

DESCRIPTION This routine receives the oldest of the highest priority message from the message queue specified by *mqdes*, subject to a timeout specified by the absolute time *abs_timeout*. 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_timedreceive( )** 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, **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 that has been waiting the longest 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, no message is removed from the queue, and **mq_receive( )** returns an error.

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.

2

EINTR
Signal received while blocking on the message queue.

ETIMEDOUT
O_NONBLOCK was not set in the message queue description associated with *mqdes*, but no message arrived on the queue before the specified timeout.

SEE ALSO **mqPxLib**, **mq_timedsend( )**

mq_timedsend( )

NAME **mq_timedsend( )** – send a message to a message queue with timeout (POSIX)

SYNOPSIS

```
int mq_timedsend
    (
    mqd_t                    mqdes,       /* message queue descriptor */
    const char *             pMsg,        /* message to send */
    size_t                   msgLen,      /* size of message, in bytes */
    unsigned                 msgPrio,     /* priority of message */
    const struct timespec *  abs_timeout  /* timeout to wait for */
    )
```

DESCRIPTION This routine adds the message *pMsg* to the message queue *mqdes* timing out if the queue is full and the message could not be sent till the absolute time specified by *abs_timeout* has passed. 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_timedsend( )** will fail.

If the message queue is not full, **mq_timedsend( )** 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's, **mq_timedsend( )** will block until the absolute time specified by *abs_timeout* has passed, or until it is interrupted by a signal. If the message queue is full and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, the message is not queued, and **mq_timedsend( )** returns with the error **EAGAIN**.

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

ETIMEDOUT
O_NONBLOCK was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

SEE ALSO **mqPxLib**, **mq_timedreceive( )**

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.

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

ENAMETOOLONG
The message queue name is exceeds _VX_PX_MQ_PATH_MAX or _VX_PX_MQ_NAME_MAX.

SEE ALSO **mqPxLib**, **mq_close( )**, **mq_open( )**

msgQClose( )

NAME

msgQClose( ) – close a named message queue

SYNOPSIS

```
STATUS msgQClose
    (
    MSG_Q_ID msgQId  /* semaphore 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.

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.

SEE ALSO

msgQLib, **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
Queue pended tasks in FIFO order.

MSG_Q_PRIORITY
Queue pended tasks in priority order.

MSG_Q_EVENTSEND_ERR_NOTIFY
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 accordingly. This option is off by default.

MSG_Q_INTERRUPTIBLE
Signal sent to a 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 is off by default.

RETURNS
The **MSG_Q_ID** of the created message queue, or **NULL** if unsuccessful.

ERRNO
S_memLib_NOT_ENOUGH_MEMORY
Not enough memory was available to allocate the required amount.

S_msgQLib_ILLEGAL_OPTIONS
An option bit other than the options described above was specified.

S_msgQLib_INVALID_MSG_LENGTH
Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT
Negative maxMsgs specified.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the message queue handle.

ENOSYS
The component **INCLUDE_MSG_Q** has not been configured into the kernel

SEE ALSO
msgQLib

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, will unblock and return **ERROR**. When this function returns, *msgQId* is no longer a valid message queue ID.

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

ERRNO

S_objLib_OBJ_ID_ERROR
The message queue ID is invalid

S_objLib_OBJ_OPERATION_UNSUPPORTED
Deleting a named message queue is not permitted

SEE ALSO **msgQLib**

msgQEvStart( )

NAME **msgQEvStart( )** – start 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 will prevent the task from ever receiving the events wanted from the message queue.

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.

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 **msgQLibCommon.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      */
    } 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.

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.

The capability to obtain a list of task IDs of tasks blocked on the message queue is not supported from user space. Also, the ability to obtain a list of pointers to the messages queued is not supported from user space.

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.

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 is used for debugging purposes only.

RETURNS

OK or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
Invalid message queue ID.

SEE ALSO

msgQInfo, **msgQShow** (kernel)

---

msgQNumMsgs( )

NAME

msgQNumMsgs( ) – get the number of messages queued to a message queue

SYNOPSIS

```
int msgQNumMsgs
    (
    MSG_Q_ID msgQId  /* message queue to examine */
    )
```

DESCRIPTION This routine returns the number of messages currently queued to a specified message queue.

RETURNS The number of messages queued, or **ERROR**.

ERRNO **S_objLib_OBJ_ID_ERROR**
Invalid message queue ID.

SEE ALSO **msgQLib**, N/A

---

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 it searchs the name space and returns the **MSG_Q_ID** of an existing message queue with *name*. If none is found, it creates a new message queue with *name* according to 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 first be removed from the name space, and then 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

This parameter specifies the message queue object management attribute bits as follows:

OM_CREATE
Create a new message queue if a matching message queue name is not found.

OM_EXCL
When set jointly with **OM_CREATE**, create a new message queue immediately without attempting to open an existing message queue. An error condition is returned if a message queue 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 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.

RETURNS

The **MSG_Q_ID** of the opened message queue, or **NULL** if unsuccessful.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP to create the message queue.

S_msgQLib_ILLEGAL_OPTIONS
An option bit other than the options described in **msgQCreate( )** was specified.

S_msgQLib_INVALID_MSG_LENGTH
Negative maxMsgLength specified.

S_msgQLib_INVALID_MSG_COUNT
Negative maxMsgs specified.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the message queue handle.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument. *name* buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access

is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The operation attempted to create an unamed public message queue.

S_objLib_OBJ_NAME_CLASH
Both the **OM_CREATE** and **OM_EXCL** flags were set in the *mode* argument and a message queue with *name* already exists.

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.

ENOSYS
The component **INCLUDE_MSG_Q** has not been configured into the kernel

SEE ALSO **msgQLib**, **msgQCreate( )**, **msgQClose( )**, **msgQUnlink( )**

---

msgQReceive( )

NAME **msgQReceive( )** – receive a message from a message queue (system call)

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.

RETURNS The number of bytes copied to *buffer*, or **ERROR**.

ERRNO

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
The **NO_WAIT** timeout was specified but no message was available.

S_objLib_OBJ_TIMEOUT
A timeout occurred while waiting for a message.

S_objLib_OBJ_INVALID_ARGUMENT
buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be written due to access control.

S_msgQLib_INVALID_MSG_LENGTH
The message length exceeds the supplied buffer size.

EINTR
Signal received while pended on the message queue

ENOSYS
The component **INCLUDE_MSG_Q** has not been configured into the kernel

SEE ALSO **msgQLib**

msgQSend( )

NAME **msgQSend( )** – send a message to a message queue (system call)

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 free space 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.

RETURNS

OK on success or **ERROR** otherwise.

ERRNO

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
: The **NO_WAIT** timeout was specified but no free buffer space was available.

S_objLib_OBJ_INVALID_ARGUMENT
: *buffer* is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

S_objLib_OBJ_TIMEOUT
: A timeout occurred while waiting for buffer space.

S_msgQLib_INVALID_MSG_LENGTH
: THe message length exceeds the limit.

S_eventLib_EVENTSEND_FAILED
: The message queue failed to send events to the registered task. This **errno** value only occurs if the message queue was created with the **MSG_Q_EVENTSEND_ERR_NOTIFY** option.

S_msgQLib_ILLEGAL_PRIORITY
: A priority other than **MSG_PRI_NORMAL** or **MSG_PRI_URGENT** was specified.

EINTR
Signal received while pended on the message queue

ENOSYS
The component **INCLUDE_MSG_Q** has not been configured into the kernel

SEE ALSO **msgQLib**, **msgQEvStart( )**

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 already no outstanding **msgQOpen( )** call, the message queue is deleted. 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.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL**. *name* buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

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 found while deleting the message queue.

SEE ALSO **msgQLib**, **msgQOpen( )**, **msgQClose( )**

msync( )

NAME

msync( ) – synchronize a file with a physical storage

SYNOPSIS

```
int msync
    (
    void * addr,  /* address to memory block */
    size_t len,   /* size of memory block */
    int    flags  /* sync/async/invalidate flags */
    )
```

DESCRIPTION

The **msync( )** function synchronizes data for memory mapped files to the permanent storage locations, in those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes.

The *addr* parameter must be a multiple of the page size as returned by **sysconf( )**.

For mappings to files, the **msync( )** function ensures that all write operations are completed as defined for synchronized I/O data integrity completion.

When the **msync( )** function is called on **MAP_PRIVATE** mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes.

For shared memory objects **msync( )** has no effect.

If the mapping was not established with by a call to **mmap( )**, **msync( )** returns -1.

The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following flags defined in the *sys/mman.h* header:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When **MS_ASYNC** is specified, **msync( )** returns immediately once all the write operations are initiated or queued for servicing; when **MS_SYNC** is specified, **msync( )** returns when all write operations are completed. Either **MS_ASYNC** or **MS_SYNC** is specified, but not both.

When **MS_INVALIDATE** is specified, **msync( )** ensures that all subsequent references to all copies of the mapped file are consistent with the new data.

When **msync( )** fails due to some of the pages not being mapped, then a subset of the pages may get synchronized. For example, if three pages are to be synchronized and the second page is not currently mapped, then the first may get updated, and the last page may not.

RETURNS

0 on success, or -1 in case of failure.

ERRNO

EINVAL
The value of *flags* is invalid, or the value of *addr* is not a multiple of the page size.

ENOMEM
The addresses in the range starting at addr and continuing for len bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

SEE ALSO

mmanLib, **mmap( )**, **munmap( )**

munlock( )

NAME

munlock( ) – unlock specified pages

SYNOPSIS

```
int munlock
    (
    const void * addr,   /* address to memory block */
    size_t       len     /* size of memory block */
    )
```

DESCRIPTION

This routines unlocks memory previously locked with **mlock( )**. In VxWorks paging is not implemented, therefore all mapped pages are always memory resident. Therefore this routine only validates parameters, but has no effect on the mapped memory.

RETURNS

0, or -1 if the memory does not belong to the process.

ERRNO

ENOMEM
Some or all of the address range specified by the addr and len arguments do not correspond to valid mapped pages in the address space of the process.

SEE ALSO

mmanLib, **mlock( )**, **mmap( )**

munlockall( )

NAME

munlockall( ) – unlock all pages used by a process

SYNOPSIS

```
int munlockall (void)
```

DESCRIPTION This routine unlocks all pages used by a process from being memory resident. In VxWorks memory is never paged, therefore all mapped pages are always memory resident. Therefore this routine does nothing.

RETURNS 0 always.

ERRNO N/A

SEE ALSO **mmanLib**, **mlockall( )**, **mmap( )**

---

munmap( )

NAME **munmap( )** – unmap pages of memory (syscall)

SYNOPSIS

```
int munmap
    (
    void *        addr,             /* address to unmap */
    size_t        len               /* size of block to unmap */
    )
```

DESCRIPTION This routine removes the mappings for pages in the range bounded by *addr* and *addr+len*. When the system include MMU support, further references to these pages shall result in the generation of a SIGSEGV signal to the process. **munmap( )** only has effect on pages that were mapped with **mmap( )**.

The *addr* parameter must be a multiple of the MMU page size as returned by **sysconf( )** when passed **_SC_PAGESIZE** or **_SC_PAGE_SIZE**. The *len* parameter must not be 0, but need not be page aligned. When it is not aligned, the unmapped range is is rounded to whole pages.

When **munmap( )** fails due to some of the pages not being mapped, then a subset of the pages may get unmapped. For example, if three pages are to be unmapped and the second page is not currently mapped, then the first page may get unmapped and the last page may not get unmapped.

RETURNS 0 on success, or -1 in case of failure.

ERRNO

EINVAL
addr is not page aligned, or *len* is 0, or the addresses in the range [*addr*,*addr+len*) are outside the valid range for the address space of the process.

SEE ALSO **mmanLib**, **mmap( )**, **mprotect( )**

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

EINTR
The call was interrupted by a signal.

EINVAL
The *rqtp* argument specified is less than or equal to 0 or greater than or equal to 1000 million.

SEE ALSO

timerLib, **sleep( )**, **taskDelay( )**

objDelete( )

NAME

objDelete( ) – generic object delete/close routine (system call)

SYNOPSIS

```
STATUS objDelete
    (
    OBJ_HANDLE    handle,
    int           options
    )
```

DESCRIPTION

The **objDelete( )** system call deletes or closes the WIND object referenced by *handle*, depending on the value set in *options*. If *options* is 0 (zero), the destroy routine referenced by the WIND object's class, is called. The following is a description of additional supported options:

VX_OBJ_DELETE_TASK_FORCE
If *handle* references a task, the routine **taskDeleteForce( )** is called

VX_OBJ_DELETE_CLOSE
handle is closed.

WARNING

It is not recommended the direct use of this system call to delete an object, because all the local resources (allocated memory) associated with it are not reclaimed. The preferred method is to call the library specific deletion routine (e.g. **semDelete( )**)

RETURNS

OK if the requested operation completes successfully, otherwise **ERROR**.

ERRNO

S_taskLib_ILLEGAL_OPERATION
The object referenced by *handle* is not a task and *options* is **VX_OBJ_DELETE_TASK_FORCE**.

SEE ALSO

objLib

objInfoGet( )

NAME

objInfoGet( ) – generic object information retrieve routine (system call)

SYNOPSIS

```
STATUS objInfoGet
    (
    OBJ_HANDLE        handle,
    void *            pInfo,
    UINT *            pInfoSize,
    int               level
    )
```

DESCRIPTION

The **objInfoGet( )** system call retrieves information specific to a WIND object. The *pInfo* parameter is a pointer to a place where to store the retrieved information, and *pInfoSize* is the size in bytes of that place. The following is a description of the supported values for *level*:

VX_OBJ_INFO_GET_TASK_NAME
If *handle* references a task, then its name is copied into the place pointed by *pInfo*.

VX_OBJ_INFO_GET_TASK_DESC
If *handle* references a task, then its task descriptor (**TASK_DESC**) is copied into the storage place pointed by *pInfo*.

VX_OBJ_INFO_GET_TASK_SUSPENDED
If *handle* references a task, then **TRUE** is written in the place pointed by *pInfo* if the task's state is suspended. Otherwise, **FALSE** is written.

VX_OBJ_INFO_GET_TASK_READY
If *handle* references a task, then **TRUE** is written in the place pointed by *pInfo* if the task's state is ready. Otherwise, **FALSE** is written.

VX_OBJ_INFO_GET_TASK_PENDED
If *handle* references a task, then **TRUE** is written in the place pointed by *pInfo* if the task's state is pended. Otherwise, **FALSE** is written.

VX_OBJ_INFO_GET_MSGQ_DESC
If *handle* references a message queue, then its message queue descriptor (**MSG_Q_INFO**) is copied into the storage place pointed by *pInfo*.

VX_OBJ_INFO_GET_SEM_DESC
If *handle* references a WIND semaphore, then its semaphore descriptor (**SEM_INFO**) is copied into the storage place pointed by *pInfo*.

RETURNS **OK** if the requested operation completes successfully, otherwise **ERROR**.

ERRNO **S_objLib_OBJ_INVALID_ARGUMENT**
pInfo is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden; Or it does belong to this RTP task but can not be written due to access control. *pInfoSize* is not valid in memory address; Or valid but it does not belong to this RTP task. Or it does belong to this calling RTP task but the needed accesses, both read and write, are not allowed.

SEE ALSO **objLib**

objUnlink( )

NAME **objUnlink( )** – unlink an object (system call)

SYNOPSIS

```
STATUS objUnlink
    (
    const char * name,
    enum windObjClassType classType
    )
```

DESCRIPTION This routine removes an object from the name space, and marks it as ready for deletion on the last **xxxClose( )**. In case there are already no outstanding **xxxOpen( )** calls, the object is deleted. After an object is unlinked, subsequent calls to **xxxOpen( )** using *name* will not be able to find the object, even if it has not been deleted yet. Instead, a new object could be created if **xxxOpen( )** is called with the **OM_CREATE** flag.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL**. *name* buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

S_objLib_OBJ_NOT_FOUND
No object with *name* was found.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Object is not named.

S_objLib_OBJ_DESTROY_ERROR
Error while deleting the object.

SEE ALSO

objLib

open( )

NAME

open( ) – open a file

SYNOPSIS

```
int open
    (
    const char *  name,   /* name of the file to open     */
    int           flags,  /* access control flag          */
    ...
    )
```

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 an optional argument.

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.

2

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 third optional parameter. 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( )**

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( )**.

RETURNS The number of characters output, not including the **NULL** terminator.

ERRNO Not Available

SEE ALSO **fioLib**, **printf( )**

pathconf( )

NAME **pathconf( )** – determine the current value of a configurable limit

SYNOPSIS

```
long pathconf
   (
    const char *path,     /* path name of the file to query  */
    int name              /* name represents the variable to be queried  */
   )
```

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

SYNOPSIS

```
int pause (void)
```

DESCRIPTION This routine suspends the task until delivery of a signal whose action is either to execute a signal handler or to terminate the process. If the action is to terminate the process, **pause( )** shall not return. If the action is to execute a signal handler, **pause( )** shall return after the signal handler returns.

This is a POSIX specified routine.

NOTE Since the **pause( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**
A signal is caught by the calling process.

SEE ALSO **sigLib**

pipeDevCreate( )

NAME **pipeDevCreate( )** – create a named pipe device (syscall)

SYNOPSIS

```
STATUS pipeDevCreate
    (
    const char * name,
    int nMessages,
    int nBytes
    )
```

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

S_ioLib_NO_DRIVER
The driver is not initialized.

S_intLib_NOT_ISR_CALLABLE
This function cannot be called from an ISR.

ENOSYS
The component **INCLUDE_PIPES** has not been configured into the kernel.

SEE ALSO

ioLib

pipeDevDelete( )

NAME

pipeDevDelete( ) – delete a named pipe device (syscall)

SYNOPSIS

```
STATUS pipeDevDelete
    (
    const char * name,
    BOOL         force
    )
```

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 if used indiscriminately. Use only as a last resort.

RETURNS

OK, or **ERROR** if the call fails.

ERRNO

S_ioLib_NO_DRIVER
The **pipeDrv** driver is not initialized.

S_intLib_NOT_ISR_CALLABLE
This function cannot be called from an ISR.

EMFILE
This pipe still has other open files.

EBUSY
The pipe is selected by at least one pending task.

ENOSYS
The component **INCLUDE_PIPES** has not been configured into the kernel.

SEE ALSO **ioLib**

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.

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.

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.

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( )**

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( )**

posix_trace_attr_destroy( )

NAME

posix_trace_attr_destroy( ) – destroy POSIX trace attributes structure

SYNOPSIS

```
int posix_trace_attr_destroy
    (
    trace_attr_t * pAttr  /* trace attributes structure to destroy */
    )
```

DESCRIPTION

Destroy a trace attributes object. The object is invalidated, and cannot be reused.

RETURNS

0 indicating success, or **EINVAL** if pAttr is invalid

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_attr_init( )**

posix_trace_attr_getclockres( )

NAME

posix_trace_attr_getclockres( ) – copy clock resolution from trace attributes

SYNOPSIS

```
int  posix_trace_attr_getclockres
    (
    const trace_attr_t * pAttr,     /* attributes object to read */
    struct timespec *    pTimespec  /* result */
    )
```

DESCRIPTION Read the clock resolution from a trace attributes object. The result will the smallest time interval which can be represented by the clock source

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getcreatetime( )**, **posix_trace_attr_getgenversion( )**, **posix_trace_attr_getname( )**, **posix_trace_attr_setname( )**

posix_trace_attr_getcreatetime( )

NAME **posix_trace_attr_getcreatetime( )** – copy stream creation time to struct timespec

SYNOPSIS

```
int  posix_trace_attr_getcreatetime
    (
    const trace_attr_t * pAttr,     /* attribues to use */
    struct timespec *    pTimespec  /* pointer to result */
    )
```

DESCRIPTION Read the stream creation time from the trace_attributes object

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**

posix_trace_attr_getgenversion( )

NAME **posix_trace_attr_getgenversion( )** – copy generation version from trace attributes

SYNOPSIS

```
int posix_trace_attr_getgenversion
    (
    const trace_attr_t * pAttr,      /* attributes to use */
    char *               genversion  /* pointer to result */
    )
```

DESCRIPTION Read the generation-version attribute from the attributes object into a character array. The array must have sufficient space for **TRACE_NAME_MAX** characters.

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getclockres( )**, **posix_trace_attr_getcreatetime( )**, **posix_trace_attr_getname( )**, **posix_trace_attr_setname( )**

posix_trace_attr_getlogfullpolicy( )

NAME **posix_trace_attr_getlogfullpolicy( )** – get log full policy from trace attributes

SYNOPSIS

```
int posix_trace_attr_getlogfullpolicy
    (
    const trace_attr_t *_Restrict pAttr,      /* attributes to use */
    int *_Restrict            pLogpolicy  /* pointer to result */
    )
```

DESCRIPTION Read the log full policy in force from the trace attributes structure. This will be one of **POSIX_TRACE_LOOP**, **POSIX_TRACE_UNTIL_FULL**, or **POSIX_TRACE_APPEND**.

RETURNS 0 indicating success, **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getstreamfullpolicy( )**, **posix_trace_attr_setlogfullpolicy( )**, **posix_trace_attr_setstreamfullpolicy( )**

posix_trace_attr_getlogsize( )

NAME **posix_trace_attr_getlogsize( )** – retrieve the size of the log for events

SYNOPSIS

```
int posix_trace_attr_getlogsize
    (
    const trace_attr_t *_Restrict pAttr,    /* attributes to use */
    size_t *_Restrict         pLogsize  /* pointer to result */
    )
```

DESCRIPTION Read the log size, in bytes, from the trace_attributes object. This is the maximum number of bytes available for storing user and system event data, and is only valid if the log full policy is **POSIX_TRACE_LOOP** or **POSIX_TRACE_UNTIL_FULL**

RETURNS 0 indicating success, or **EINVAL** if parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_getmaxdatasize( )

NAME **posix_trace_attr_getmaxdatasize( )** – get the maximum data size for an event

SYNOPSIS

```
int posix_trace_attr_getmaxdatasize
    (
    const trace_attr_t *_Restrict attr,        /* attributes to use */
    size_t *_Restrict         maxdatasize  /* pointer to result */
    )
```

DESCRIPTION Read the maximum allowed size of user event data, in bytes, from the trace attributes object.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_getmaxsystemeventsize( )

NAME **posix_trace_attr_getmaxsystemeventsize( )** – get maximum size of a system event

SYNOPSIS

```
int posix_trace_attr_getmaxsystemeventsize
    (
    const trace_attr_t *_Restrict attr,      /* attributes to use */
    size_t *_Restrict         eventsize  /* size to set */
    )
```

DESCRIPTION Calculate the maximum size, in bytes, required to store a system trace event.

RETURNS

0 indicating success, or **EINVAL** for invalid arguments

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_getmaxusereventsize( )

NAME

posix_trace_attr_getmaxusereventsize( ) – get the maximum size of user event

SYNOPSIS

```
int posix_trace_attr_getmaxusereventsize
    (
    const trace_attr_t *_Restrict attr,       /* attrributes to use */
    size_t                       data_len,  /* size of user data */
    size_t *_Restrict            eventsize  /* pointer to result */
    )
```

DESCRIPTION

Calculate the memory requirement for storing a user trace event with the supplied *data_len* parameter. The current setting for the maxdatasize property of the trace attributes object is taken into consideration.

RETURNS

0 indicating success, or **EINVAL** for invalid arguments

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_getname( )

NAME

posix_trace_attr_getname( ) – copy stream name from trace attributes

SYNOPSIS

```
int posix_trace_attr_getname
    (
    const trace_attr_t * pAttr,     /* attributes to use */
```

```
    char *              tracename  /* pointer to result */
    )
```

DESCRIPTION Copy the trace name from the attributes object to a character array. The array must have space for **TRACE_NAME_MAX** characters.

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getclockres( )**, **posix_trace_attr_getcreatetime( )**, **posix_trace_attr_getgenversion( )**, **posix_trace_attr_setname( )**

posix_trace_attr_getstreamfullpolicy( )

NAME **posix_trace_attr_getstreamfullpolicy( )** – get stream full policy

SYNOPSIS

```
int posix_trace_attr_getstreamfullpolicy
    (
    const trace_attr_t * pAttr,          /* attribues to use */
    int *               pStreampolicy  /* pointer to result */
    )
```

DESCRIPTION Read the stream full policy in force from the trace attributes object. This may be **POSIX_TRACE_LOOP**, **POSIX_TRACE_FULL**, or (for streams with log only) **POSIX_TRACE_FLUSH**.

RETURNS 0 indicating success, or **EINVAL** if parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogfullpolicy( )**, **posix_trace_attr_setlogfullpolicy( )**, **posix_trace_attr_setstreamfullpolicy( )**

posix_trace_attr_getstreamsize( )

NAME **posix_trace_attr_getstreamsize( )** – get the size of memory used for event data

SYNOPSIS

```
int posix_trace_attr_getstreamsize
    (
    const trace_attr_t *_Restrict pAttr,       /* attributes to use */
```

```
    size_t *_Restrict         pStreamsize  /* pointer to result */
    )
```

DESCRIPTION Get the stream size, in bytes, from the trace attributes object. The stream size is the total memory to be used for storing user and system event data.

RETURNS 0 indicating success, or **EINVAL** if parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_init( )

NAME **posix_trace_attr_init( )** – initialize a POSIX trace attributes structure

SYNOPSIS

```
int posix_trace_attr_init
    (
    trace_attr_t * pAttr  /* address of structure to initialize */
    )
```

DESCRIPTION Initialize a trace attributes object with default values. Required properties can then be set on this structure before creating a trace.

RETURNS 0 indicating success, **EINVAL** if pAttr is **NULL**

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_destroy( )**

posix_trace_attr_setlogfullpolicy( )

NAME **posix_trace_attr_setlogfullpolicy( )** – set log full policy in trace attributes

SYNOPSIS

```
int posix_trace_attr_setlogfullpolicy
    (
    trace_attr_t * pAttr,     /* attributes to update */
```

```
    int             logpolicy  /* policy to apply */
    )
```

DESCRIPTION Set the log full policy in the trace_attr_t structure. The policy must be one of **POSIX_TRACE_LOOP**, **POSIX_TRACE_UNTIL_FULL**, or **POSIX_TRACE_APPEND**.

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogfullpolicy( )**, **posix_trace_attr_getstreamfullpolicy( )**, **posix_trace_attr_setstreamfullpolicy( )**

---

posix_trace_attr_setlogsize( )

NAME **posix_trace_attr_setlogsize( )** – set the size of event data in a log

SYNOPSIS

```
int posix_trace_attr_setlogsize
    (
    trace_attr_t * pAttr,   /* attributes to update */
    size_t         logsize  /* size to apply */
    )
```

DESCRIPTION Set the maximum allowed size, in bytes, of the event data stored in the log. This is ignored if the log full policy is **POSIX_TRACE_APPEND**

RETURNS 0 indicating success, or **EINVAL** if parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_setstreamsize( )**

---

posix_trace_attr_setmaxdatasize( )

NAME **posix_trace_attr_setmaxdatasize( )** – set the maximum user event data size

SYNOPSIS

```
int posix_trace_attr_setmaxdatasize
```

```
    (
    trace_attr_t * pAttr,       /* attributes to update */
    size_t         maxdatasize  /* size to use */
    )
```

DESCRIPTION Set the maximum allowed size for the user data which may be passed to **posix_trace_event( )**. Data longer that this will be truncated.

RETURNS 0 indicating success, or **EINVAL** if attributes are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_getstreamsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setstreamsize( )**

posix_trace_attr_setname( )

NAME **posix_trace_attr_setname( )** – set the stream name in trace attributes

SYNOPSIS

```
int posix_trace_attr_setname
    (
    trace_attr_t * pAttr,     /* attributes to update */
    const char *   tracename  /* name to give to stream */
    )
```

DESCRIPTION Set the name in the trace attributes object. If the supplied name is greater than **TRACE_NAME_MAX** characters in length, it will be truncated to (**TRACE_NAME_MAX** - 1) characters, and a NUL appended.

RETURNS 0 indicating success, or **EINVAL** if the parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getclockres( )**, **posix_trace_attr_getcreatetime( )**, **posix_trace_attr_getgenversion( )**, **posix_trace_attr_setname( )**

posix_trace_attr_setstreamfullpolicy( )

NAME **posix_trace_attr_setstreamfullpolicy( )** – set stream full policy

SYNOPSIS

```
int posix_trace_attr_setstreamfullpolicy
    (
    trace_attr_t * pAttr,        /* attributes to update */
    int            streampolicy  /* policy to apply */
    )
```

DESCRIPTION Set the stream full policy in a trace attributes object. This may be **POSIX_TRACE_LOOP**, **POSIX_TRACE_FULL**, or (for streams with log only) **POSIX_TRACE_FLUSH**.

RETURNS 0 indicating success, or **EINVAL** if parameters are invalid

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogfullpolicy( )**, **posix_trace_attr_getstreamfullpolicy( )**, **posix_trace_attr_setlogfullpolicy( )**

---

posix_trace_attr_setstreamsize( )

NAME **posix_trace_attr_setstreamsize( )** – set size of memory to be used for event data

SYNOPSIS

```
int posix_trace_attr_setstreamsize
    (
    trace_attr_t * pAttr,      /* attributes to update */
    size_t         streamsize  /* size to set */
    )
```

DESCRIPTION Set the stream size, in bytes, in the trace attributes object. The stream size is the total memory to be used for storing user and system event data only, and does not include other overhead.

RETURNS 0 indicating success, or **EINVAL** for invalid attributes

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_attr_getlogsize( )**, **posix_trace_attr_getmaxdatasize( )**, **posix_trace_attr_getmaxsystemeventsize( )**, **posix_trace_attr_getmaxusereventsize( )**, **posix_trace_attr_setlogsize( )**, **posix_trace_attr_setmaxdatasize( )**, **posix_trace_attr_getstreamsize( )**

---

posix_trace_clear( )

NAME **posix_trace_clear( )** – reinitialize a trace stream

SYNOPSIS

```
int posix_trace_clear
    (
    trace_id_t trid  /* trace stream to clear */
    )
```

DESCRIPTION Reinitialize the trace stream as though just created, but reuse the allocated resources. The eventname mappings are unchanged, and if running, the trace status remains running. Not all file types can support this operation, as it requires a seek in the file.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**

posix_trace_close( )

NAME **posix_trace_close( )** – close a pre-recorded trace stream

SYNOPSIS

```
int posix_trace_close
    (
    trace_id_t trid  /* trace stream to close */
    )
```

DESCRIPTION Close the trace log associated with the supplied trace id. All the resources used by the trace will be released.

RETURNS 0, or **EINVAL** for an invalid trace id

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_open( )**, **posix_trace_rewind( )**

posix_trace_create( )

NAME **posix_trace_create( )** – create a trace stream without a log

SYNOPSIS

```
int  posix_trace_create
    (
    pid_t                     pid,    /* process to trace */
    const trace_attr_t *_Restrict pAttr,  /* trace attributes to use */
```

```
    trace_id_t *_Restrict     pTrid   /* pointer to created trace */
    )
```

DESCRIPTION Create a POSIX trace using the supplied trace attributes object. Tracing is not active until **posix_trace_start( )** is called.

If the *pid* parameter is 0, then the current process will be traced. If *pAttr* is **NULL**, then default values will be used.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_create_withlog( )**, **posix_trace_shutdown( )**, **posix_trace_flush( )**

posix_trace_create_withlog( )

NAME **posix_trace_create_withlog( )** – create a trace stream with a log file

SYNOPSIS

```
int posix_trace_create_withlog
    (
    pid_t                     pid,    /* process to trace */
    const trace_attr_t *_Restrict pAttr,  /* attributes to use */
    int                       fd,     /* descriptor for log file */
    trace_id_t *_Restrict     pTrid   /* pointer to resultin trace */
    )
```

DESCRIPTION Create a POSIX trace using the supplied trace attributes object and file descriptor. This function is equivalent to **posix_trace_create( )** but also associates a trace log with the stream. Tracing is not active until **posix_trace_start( )** is called.

If the *pid* parameter is 0, then the current process will be traced. If *pAttr* is **NULL**, then defaults will be used.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_create( )**, **posix_trace_shutdown( )**, **posix_trace_flush( )**

posix_trace_event( )

NAME

posix_trace_event( ) – record an event

SYNOPSIS

```
void posix_trace_event
   (
   trace_event_id_t           event_id,       /* event to store */
   const void * _Restrict     data_ptr,       /* pointer to event data */
   size_t                     data_len        /* size of event data */
   )
```

DESCRIPTION

Record an event from a process into an active trace stream, with user-supplied data. The user-supplied data is passed as a pointer and a length. Not all of the data may be stored: It it exceeds the value set by **posix_trace_attr_setmaxdatasize( )** then the data will be truncated in the trace, and the event truncation status set to **POSIX_TRACE_TRUNCATED_RECORD**

RETURNS

n/a

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_eventid_open( )**

posix_trace_eventid_equal( )

NAME

posix_trace_eventid_equal( ) – compare two event ids

SYNOPSIS

```
int  posix_trace_eventid_equal
    (
    trace_id_t       trid,    /* trace stream */
    trace_event_id_t event1,  /* first event id */
    trace_event_id_t event2   /* second event id */
    )
```

DESCRIPTION

Compare two event ids from a trace stream.

RETURNS

0 if the event ids are equal, non-zero otherwise

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_eventid_getname( )**, **posix_trace_trid_eventid_open( )**

posix_trace_eventid_get_name( )

NAME **posix_trace_eventid_get_name( )** – retrieve the name for a POSIX event id

SYNOPSIS

```
int posix_trace_eventid_get_name
    (
    trace_id_t       trid,        /* trace stream */
    trace_event_id_t event,       /* event id to find name of */
    char *           event_name   /* pointer to result array */
    )
```

DESCRIPTION For a given event id, look up its name in the list of event types in the stream. The name will be written into the character array *event_name*, which must have sufficient space for not less than **TRACE_EVENT_NAME_MAX** characters. If the name could not be found, return **EINVAL**.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**

posix_trace_eventid_open( )

NAME **posix_trace_eventid_open( )** – retrieve an event id for the supplied name

SYNOPSIS

```
int posix_trace_eventid_open
    (
    const char *_Restrict     event_name,  /* name of event */
    trace_event_id_t *_Restrict event_id     /* address of numeric result */
    )
```

DESCRIPTION Get an event id to associate with a named user event. When passed a string representing an event name, this function will provide an event id. If the string is already associated with an id, that id will be returned.

If all the available events have been used, the id **POSIX_TRACE_UNNAMED_USEREVENT** will be used.

RETURNS 0 indicating success

ENAMETOOLONG if the supplied string is greater than **TRACE_EVENT_NAME_MAX** in length.

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_event( )**

posix_trace_eventset_add( )

NAME **posix_trace_eventset_add( )** – add a POSIX trace event id to an event set

SYNOPSIS

```
int posix_trace_eventset_add
    (
    trace_event_id_t    event_id,  /* eventId to add to the set */
    trace_event_set_t * set        /* event set to be modified */
    )
```

DESCRIPTION Add a specified event to an event set. Applications must call either **posix_trace_eventset_empty( )** or **posix_trace_eventset_fill( )** before performing any other operations on the set.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_eventset_del( )**, **posix_trace_eventset_empty( )**, **posix_trace_eventset_fill( )**, **posix_trace_eventset_ismember( )**

posix_trace_eventset_del( )

NAME **posix_trace_eventset_del( )** – remove a POSIX trace event id from an event set

SYNOPSIS

```
int posix_trace_eventset_del
    (
    trace_event_id_t    event_id,  /* eventId to add to the set */
    trace_event_set_t * set        /* event set to be modified */
    )
```

DESCRIPTION Remove a specified event from an event set. Applications must call either **posix_trace_eventset_empty( )** or **posix_trace_eventset_fill( )** before performing any other operations on the set.

RETURNS
0 indicating success
EINVAL if eventset is invalid or uniniatialized

ERRNO

SEE ALSO
pxTraceLib, **posix_trace_eventset_add( )**, **posix_trace_eventset_empty( )**, **posix_trace_eventset_fill( )**, **posix_trace_eventset_ismember( )**

posix_trace_eventset_empty( )

NAME
posix_trace_eventset_empty( ) – remove all events from an event set

SYNOPSIS
```
int posix_trace_eventset_empty
    (
    trace_event_set_t * pSet  /* eventset to empty */
    )
```

DESCRIPTION
Remove all events from the event set pointed to by *pSet*. Applications must call either **posix_trace_eventset_empty( )** or **posix_trace_eventset_fill( )** before performing any other operations on the set.

RETURNS
0 indicating success
EINVAL if eventset is invalid

ERRNO

SEE ALSO
pxTraceLib, **posix_trace_eventset_add( )**, **posix_trace_eventset_del( )**, **posix_trace_eventset_fill( )**, **posix_trace_eventset_ismember( )**

posix_trace_eventset_fill( )

NAME
posix_trace_eventset_fill( ) – fill an event set with a set of events

SYNOPSIS
```
int posix_trace_eventset_fill
    (
    trace_event_set_t * pSet,   /* eventSet to update */
    int                 what    /* fill mode */
    )
```

DESCRIPTION

Fill an event set according to the requested mode. The mode may be **POSIX_TRACE_WOPID_EVENTS** (which adds all the process-independent events to the set), **POSIX_TRACE_SYSTEM_EVENTS** (which adds all the system events to the set), or **POSIX_TRACE_ALL_EVENTS** (which adds all events) Applications must call either **posix_trace_eventset_empty( )** or **posix_trace_eventset_fill( )** before performing any other operations on the set.

RETURNS

0 indicating success
EINVAL if eventset or fill mode is invalid

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_eventset_add( )**, **posix_trace_eventset_del( )**, **posix_trace_eventset_empty( )**, **posix_trace_eventset_ismember( )**

---

posix_trace_eventset_ismember( )

NAME

posix_trace_eventset_ismember( ) – test whether a POSIX trace event is in a set

SYNOPSIS

```
int posix_trace_eventset_ismember
    (
    trace_event_id_t          event_id,  /* eventId to add to set */
    const trace_event_set_t *_Restrict set,       /* event set to be tested
*/
    int *_Restrict            isMember   /* result */
    )
```

DESCRIPTION

Test a specified event for membership of an event set. If the event is a member of the set, the variable pointed to by *isMember* will be non-zero, otherwise, it will be zero. Applications must call either **posix_trace_eventset_empty( )** or **posix_trace_eventset_fill( )** before performing any other operations on the set.

RETURNS

0 indicating success
EINVAL if eventset is invalid or uniniatialized

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_eventset_add( )**, **posix_trace_eventset_del( )**, **posix_trace_eventset_fill( )**, **posix_trace_eventset_empty( )**

posix_trace_eventtypelist_getnext_id( )

NAME **posix_trace_eventtypelist_getnext_id( )** – retrieve an event id for a stream

SYNOPSIS

```
int posix_trace_eventtypelist_getnext_id
    (
    trace_id_t                trid,        /* trace stream */
    trace_event_id_t *_Restrict event,       /* pointer to result */
    int *_Restrict            unavailable  /* flag for end of list */
    )
```

DESCRIPTION When called for the first time, return, in the variable pointed to by the *event* parameter, the first trace event type identifier of the list of trace event types. Successive calls will return all the trace event types, until there are no more. After the last event id has been returned, the variable pointed to by the *unavailable* parameter will be set to non-zero. Event ids are not returned in any specific order.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_eventtypelist_rewind( )**

posix_trace_eventtypelist_rewind( )

NAME **posix_trace_eventtypelist_rewind( )** – reset the event id list iterator

SYNOPSIS

```
int posix_trace_eventtypelist_rewind
    (
    trace_id_t trid  /* trace stream */
    )
```

DESCRIPTION Reset the next trace event type identifier to be the first trace event type identifier in the list of events for the trace *trid*

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_eventtypelist_getnext_id( )**

posix_trace_flush( )

NAME

posix_trace_flush( ) – flush trace stream contents to trace log

SYNOPSIS

```
int posix_trace_flush
   (
   trace_id_t         trid       /* trace stream to flush */
   )
```

DESCRIPTION

This function flushes the contents of a trace stream to the associated trace log. If there is no log associated with the trace, the function returns **EINVAL**

RETURNS

0 indicating success, or an error number

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_create( )**, **posix_trace_create_withlog( )**, **posix_trace_shutdown( )**

posix_trace_get_attr( )

NAME

posix_trace_get_attr( ) – get the status of a trace stream

SYNOPSIS

```
int posix_trace_get_attr
    (
    trace_id_t     trid,  /* trace stream */
    trace_attr_t * attr   /* destination */
    )
```

DESCRIPTION

Read the trace attributes for the given trace object

RETURNS

0, or an error value

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_get_status( )**

posix_trace_get_filter( )

NAME

posix_trace_get_filter( ) – get the event filter set from a stream

SYNOPSIS

```
int posix_trace_get_filter
    (
    trace_id_t          trid,  /* trace stream to get filter from */
    trace_event_set_t * set    /* destination */
    )
```

DESCRIPTION Read the trace event filter from the supplied trace object

RETURNS 0, or an error value

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_set_filter( )**

posix_trace_get_status( )

NAME **posix_trace_get_status( )** – retrieve the status of a stream

SYNOPSIS

```
int posix_trace_get_status
    (
    trace_id_t                trid,       /* trace stream */
    struct posix_trace_status_info * statusinfo  /* destination */
    )
```

DESCRIPTION Read the trace status from the supplied trace object

RETURNS 0, or an error value

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_get_attr( )**

posix_trace_getnext_event( )

NAME **posix_trace_getnext_event( )** – retrieve an event from a stream

SYNOPSIS

```
int posix_trace_getnext_event
    (
    trace_id_t                trid,        /* trace stream */
    struct posix_trace_event_info *
    _Restrict                 event,       /* destination for event info */
    void *_Restrict           data,        /* destination for event data */
```

```
    size_t                    num_bytes,   /* size of event data destination
*/
    size_t *_Restrict         data_len,    /* amount of data written */
    int *_Restrict            unavailable  /* flag indicating no event
available */
    )
```

DESCRIPTION Attempt to read the next event from a trace without a log, or a pre-precorded trace. If there is no event available, the variable pointed to by *unavailable* will be set non-zero. *num_bytes* is the amount of space available for the event to be written into. On return, the variable pointed to by *data_len* will indicate the number of bytes transferred. The truncated flag in the posix_trace_event_info structure will be updated appropriately: If the event has not been truncated, the flag will be set to **POSIX_TRACE_NOT_TRUNCATED**. If the event was truncated when written, the flag will be **POSIX_TRACE_TRUNCATED**, and if truncated on read, the status will be set to **POSIX_TRUNCATED_READ**, and *data_len* will be set equal to num_bytes.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_timedgetnext_event( )**, **posix_trace_trygetnext_event( )**

posix_trace_open( )

NAME **posix_trace_open( )** – create a stream from a pre-recorded trace log

SYNOPSIS

```
int posix_trace_open
    (
    int           fd,    /* open file descriptor to read */
    trace_id_t * trid   /* pointer to resulting trace stream */
    )
```

DESCRIPTION This function allocates resources and creates a trace stream, which is associated with the supplied file descriptor. The file descriptor must be open for reading, and must be able to support **seek( )**

RETURNS 0, or **EINVAL** for an invalid trace id

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_close( )**, **posix_trace_rewind( )**

posix_trace_rewind( )

NAME **posix_trace_rewind( )** – read the next event from the start of the trace

SYNOPSIS

```
int posix_trace_rewind
    (
    trace_id_t trid  /* trace stream to rewind */
    )
```

DESCRIPTION This function resets the current trace event timestamp to that of the first event in the trace stream identified by *trid*.

RETURNS 0, or **EINVAL** for an invalid trace id

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_close( )**, **posix_trace_open( )**

posix_trace_set_filter( )

NAME **posix_trace_set_filter( )** – set the event filter associated with a stream

SYNOPSIS

```
int posix_trace_set_filter
   (
   trace_id_t                  trid,   /* id of trace to update */
   const trace_event_set_t *  set,    /* pointer to eventset */
   int                         how     /* type of modification to make */
   )
```

DESCRIPTION Apply a filter to the trace object identified by *trid*. The type of modification that is made is controlled by the *how* parameter. This may be **POSIX_TRACE_SET_EVENTSET**, **POSIX_TRACE_ADD_EVENTSET** or **POSIX_TRACE_SUB_EVENTSET**

RETURNS 0, or an error value

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_get_filter( )**

posix_trace_shutdown( )

NAME

posix_trace_shutdown( ) – stop tracing and destroy the stream

SYNOPSIS

```
int posix_trace_shutdown
   (
   trace_id_t         trid       /* trace stream to shutdown */
   )
```

DESCRIPTION

This function stops tracing. If the stream has a log associated with it, the stream is flushed, and then the stream is deleted and the file closed.

RETURNS

0 indicating success, or an error number

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_create_withlog( )**, **posix_trace_create( )**, **posix_trace_flush( )**

posix_trace_start( )

NAME

posix_trace_start( ) – start tracing using a pre-existing trace object

SYNOPSIS

```
int posix_trace_start
   (
   trace_id_t         trid       /* trace stream to start */
   )
```

DESCRIPTION

Start tracing with the supplied trace object.

RETURNS

0 indicating success, or an error number

ERRNO

SEE ALSO

pxTraceLib, **posix_trace_stop( )**

posix_trace_stop( )

NAME

posix_trace_stop( ) – stop tracing

SYNOPSIS

```
int posix_trace_stop
   (
   trace_id_t         trid       /* trace stream to stop */
   )
```

DESCRIPTION Stop tracing with the supplied trace object.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_start( )**

posix_trace_timedgetnext_event( )

NAME **posix_trace_timedgetnext_event( )** – retrieve an event from a stream, with timeout

SYNOPSIS

```
int posix_trace_timedgetnext_event
    (
    trace_id_t                trid,         /* trace stream */
    struct posix_trace_event_info *
    _Restrict                 event,        /* destination for event info */
    void *_Restrict           data,         /* destination for event data */
    size_t                   num_bytes,    /* size of event data destination
*/
    size_t *_Restrict         data_len,     /* amount of data written */
    int *_Restrict            unavailable,  /* flag indicating no data */
    const struct timespec     *
    _Restrict                 abs_timeout   /* maximum time to wait for data
*/
    )
```

DESCRIPTION Attempt to read the next event from a trace without a log, or a pre-precorded trace. If there is no event available after the specified time, the variable pointed to by the *unavailable* parameter will be set non-zero, and **ETIMEDOUT** will be returned. *num_bytes* is the amount of space available for the event to be written into. On return, if successful, the variable pointed to by *data_len* will indicate the number of bytes transferred. The truncated flag in the posix_trace_event_info structure will be updated appropriately: If the event has not been truncated, the flag will be set to **POSIX_TRACE_NOT_TRUNCATED**. If the event was truncated when written, the flag will be **POSIX_TRACE_TRUNCATED**, and if truncated on read, the flag will be set to **POSIX_TRUNCATED_READ**, and *data_len* will be set equal to *num_bytes*.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_getnext_event( )**, **posix_trace_trygetnext_event( )**

posix_trace_trid_eventid_open( )

NAME **posix_trace_trid_eventid_open( )** – retrieve an event id for the supplied name

SYNOPSIS

```
int  posix_trace_trid_eventid_open
    (
    trace_id_t                trid,         /* trace to get id from */
    const char *_Restrict     event_name,   /* name of event */
    trace_event_id_t *_Restrict event_id      /* address of numeric result */
    )
```

DESCRIPTION Get an event id to associate with a named user event for a trace. When passed a string representing an event name, this function will provide an event id. If the string is already associated with an id, that id will be returned.

If all the available events have been used, the id **POSIX_TRACE_UNNAMED_USEREVENT** will be used.

RETURNS 0 indicating success
ENAMETOOLONG if the supplied string is greater than **TRACE_EVENT_NAME_MAX** in length

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_eventid_equal( )**, **posix_trace_eventid_get_name( )**

posix_trace_trygetnext_event( )

NAME **posix_trace_trygetnext_event( )** – try to retrieve an event from a stream

SYNOPSIS

```
int posix_trace_trygetnext_event
    (
    trace_id_t                trid,         /* trace stream */
    struct posix_trace_event_info *
    _Restrict                 event,        /* destination for event info */
    void *_Restrict           data,         /* destination for event data */
    size_t                    num_bytes,    /* size of event data destination
*/
```

```
    size_t *_Restrict          data_len,    /* number of bytes written */
    int *_Restrict             unavailable  /* flag for no event available */
    )
```

DESCRIPTION Attempt to read the next event from a trace without a log. If there is no event available, the variable pointed to by *unavailable* will be set non-zero. *num_bytes* is the amount of space available for the event to be written into. On return, the variable pointed to by *data_len* will indicate the number of bytes transferred. The truncated flag in the posix_trace_event_info structure will be updated appropriately: If the event has not been truncated, the flag will be set to **POSIX_TRACE_NOT_TRUNCATED**. If the event was truncated when written, the flag will be **POSIX_TRACE_TRUNCATED**, and if truncated on read, the flag will be set to **POSIX_TRUNCATED_READ**, and *data_len* will be set equal to *num_bytes*.

RETURNS 0 indicating success, or an error number

ERRNO

SEE ALSO **pxTraceLib**, **posix_trace_timedgetnext_event( )**, **posix_trace_getnext_event( )**

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( )**.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

ERRNO Not Available

SEE ALSO **fioLib**, **printf( )**

pthread_atfork( )

NAME

pthread_atfork( ) – register fork handlers (POSIX)

SYNOPSIS

```
int pthread_atfork
    (
    void (*prepare)(void),
    void (*parent)(void),
    void (*child)(void)
    )
```

DESCRIPTION

This routine declares handlers to be called before and after **fork( )**.

WARNING

Because the **fork( )** function is not provided in VxWorks, this implementation of **pthread_atfork( )** does nothing and always returns **ERROR**.

RETURNS

ERROR always.

ERRNO

N/A

SEE ALSO

pthreadLib

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_getguardsize( )

NAME

pthread_attr_getguardsize( ) – get the thread guard size (POSIX)

SYNOPSIS

```
int pthread_attr_getguardsize
    (
    const pthread_attr_t * _Restrict pAttr,      /* thread attributes    */
    size_t * _Restrict        pGuardsize  /* size of guarded area */
    )
```

DESCRIPTION

This routine gets the guard size from the thread attributes object *pAttr* and stores it in the location pointed to by *pGuardsize*.

RETURNS

zero on success, **EINVAL** if an invalid thread attribute or invalid *pGuardsize* is passed.

ERRNO

None.

SEE ALSO

pthreadLib, **pthread_attr_setguardsize( )**, **pthread_attr_init( )**, **pthread_create( )**

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 * _Restrict pAttr,         /* thread attributes
object */
    int * _Restrict           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 non-POSIX routine gets the name in the specified thread attributes object, *pAttr*.

This routine expects the *name* parameter to be a valid storage space.

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 * _Restrict pAttr,  /* thread attributes */
    struct sched_param * _Restrict 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 * _Restrict pAttr,   /* thread attributes    */
    int * _Restrict           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 * _Restrict pAttr,            /* thread attributes
object */
```

```
    int * _Restrict           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_getstack( )

NAME

pthread_attr_getstack( ) – get stack attributes from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getstack
    (
    const pthread_attr_t * _Restrict pAttr,        /* thread attributes */
    void ** _Restrict         ppStackaddr,  /* current stack addr (out) */
    size_t * _Restrict        pStackSize    /* current stack size (out) */
    )
```

DESCRIPTION

This routine gets the stack address and stack size from the thread attributes object *pAttr* and stores them in the location pointed to by *ppStackaddr* and *pStackSize* respectively.

RETURNS

zero on success, **EINVAL** if an invalid thread attribute is passed or if *ppStackaddr* or *pStackSize* is **NULL**.

ERRNO

None.

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_setstack( )**, **pthread_attr_getstacksize( )**, **pthread_attr_setstackaddr( )**, **pthread_attr_getstackaddr( )**

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 * _Restrict pAttr,       /* thread attributes */
    void ** _Restrict         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*.

NOTE

This API has been obsoleted by the standard. The standard says "The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-2001. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features."

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 * _Restrict pAttr,      /* thread attributes */
    size_t * _Restrict        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 default stack size for POSIX threads (20480 bytes).

Guard Size
the value as returned by **VM_PAGE_SIZE_GET( )**.

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). \iP **Scheduling Priority** 127 - medium priority between minimum (0) and maximum (255).

The following default attributes are set for the **SCHED_SPORADIC** policy:

Low Scheduling Priority
63 - half of the default priority.

Replenishment Period
10 seconds.

Initial Budget
4 seconds.

Maximum Pending Replenishments
40

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 - the floating point option is set.

RETURNS On success zero; on failure the **EINVAL** error code.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_attr_destroy( )**, **pthread_attr_getdetachstate( )**, **pthread_attr_getguardsize( )**, **pthread_attr_getinheritsched( )**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_getscope( )**, **pthread_attr_getstack( )**, **pthread_attr_getstackaddr( )**, **pthread_attr_getstacksize( )**, **pthread_attr_setdetachstate( )**, **pthread_attr_setguardsize( )**, **pthread_attr_setinheritsched( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**, **pthread_attr_setscope( )**, **pthread_attr_setstack( )**, **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_setguardsize( )

NAME **pthread_attr_setguardsize( )** – set the thread guard size (POSIX)

SYNOPSIS

```
int pthread_attr_setguardsize
    (
    pthread_attr_t *pAttr,    /* thread attributes            */
    size_t         guardsize  /* guard size                   */
    )
```

DESCRIPTION This routine sets the guard size in the thread attributes object *pAttr* to *guardsize*.

If *guardsize* is zero, a guard area is not provided for threads created with *pAttr*. If guardsize is greater than zero, a guard area of at least size guardsize bytes is provided for each thread created with *pAttr*.

If the stack address and stack size attributes are set, the guard size attribute is be ignored and no protection is provided.

RETURNS zero on success, **EINVAL** if an invalid thread attribute is passed.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_getguardsize( )**, **pthread_attr_init( )**, **pthread_create( )**

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 non-POSIX routine sets the name in the specified thread attributes object, *pAttr*. This allows for specifying a non-default name for the VxWorks task acting as a thread.

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.

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( )**

pthread_attr_setschedparam( )

NAME **pthread_attr_setschedparam( )** – set schedparam attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setschedparam
    (
    pthread_attr_t * _Restrict pAttr,  /* thread attributes    */
    const struct sched_param * _Restrict pParam  /* new parameters       */
    )
```

DESCRIPTION Set the scheduling parameters in the thread attributes object *pAttr*. For all scheduling policies the common scheduling parameter is:

- the thread's execution priority.

Additionally for the **SCHED_SPORADIC** policy the following scheduling parameters have to be provided:

- the low scheduling priority.

- the replenishment period.

- the initial budget.

- the maximum pending replenishment.

If a thread priority is being set explicitly, 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_getschedpolicy( )**, **pthread_attr_setschedpolicy( )**, **pthread_attr_getinheritsched( )**, **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. If a scheduling policy is being set explicitly, the **PTHREAD_EXPLICIT_SCHED** mode must be set (see **pthread_attr_setinheritsched( )** for information) for the policy to take effect.

POSIX defines the following policies:

SCHED_RR
Realtime, round-robin scheduling.

SCHED_FIFO
Realtime, first-in first-out scheduling.

SCHED_SPORADIC
Sporadic server scheduling policy.

SCHED_OTHER
Other, active VxWorks native scheduling policy.

RETURNS
On success zero; on failure the **EINVAL** error code if the thread attribute is not valid, or not initialized, or if the policy is not valid.

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. Any other value passed to this function will result in **EINVAL** 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_setstack( )

NAME **pthread_attr_setstack( )** – set stack attributes in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setstack
    (
    pthread_attr_t *pAttr,       /* thread attributes            */
    void           *pStackaddr,  /* new stack address            */
    size_t         stacksize     /* new stack size               */
    )
```

DESCRIPTION This routine sets the stack address and stack size in the thread attributes object *pAttr* to *pStackaddr* and *stacksize*. *pStackaddr* must be the lowest address of the stack regardless of what the thread considers as the stack base or the stack end.

The memory area used as 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 exiting 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.

NOTE VxWorks currently does not check whether the stack page(s) described by *pStackaddr* and *stacksize* are both readable and writable by the thread. The POSIX standard does not mandate it.

RETURNS
zero on success, **EINVAL** if an invalid thread attribute is passed, *pStackaddr* is invalid or if *stacksize* is lower than **PTHREAD_STACK_MIN**.

ERRNO
None.

SEE ALSO
pthreadLib, **pthread_attr_getstack( )**, **pthread_attr_setstackaddr( )**, **pthread_attr_setstacksize( )**, **pthread_attr_init( )**, **pthread_create( )**

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( )**. However constraints may be imposed by other methods used to allocate the memory area used as a stack (in particular see **mmanLib** for setting up guard pages around the stack)

The memory area used as 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 exiting 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( )**.

NOTE
This API has been obsoleted by the standard. The standard says "The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-2001. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features."

RETURNS
zero on success, **EINVAL** if an invalid thread attribute is passed.

ERRNO None.

SEE ALSO **pthreadLib**, **pthread_attr_setstacksize( )**, **pthread_attr_init( )**, **pthread_create( )**

pthread_attr_setstacksize( )

NAME **pthread_attr_setstacksize( )** – set stack size 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 pthread 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**.

See also the list of cancellation points in system calls and library calls detailed in the **pthreadLib** documentation.

IMPLEMENTATION NOTES

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.

Please also note that all threads that remain *joinable* at the time they are cancelled 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 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.

The **pthread_cond_broadcast( )** function may be called by a thread whether or not it currently owns the mutex that threads calling **pthread_cond_wait( )** or **pthread_cond_timedwait( )** have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex must be locked by the thread calling **pthread_cond_broadcast( )**.

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 one of the following non-zero error code: **EINVAL**, **EBUSY**

ERRNO N/A

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 * _Restrict pCond,  /* condition variable */
    const pthread_condattr_t * _Restrict pAttr   /* condition var. 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.

NOTE this routine does not verify whether the *pCond* parameter corresponds to an already initialized condition variable object. It is up to the caller to ensure that *pCond* does not correspond to an object already in use.

RETURNS On success zero; on failure one of the following non-zero error code: **EINVAL**, **EAGAIN**

ERRNO N/A

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.

The **pthread_cond_signal( )** function may be called by a thread whether or not it currently owns the mutex that threads calling **pthread_cond_wait( )** or **pthread_cond_timedwait( )** have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex must be locked by the thread calling **pthread_cond_signal( )**.

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 * _Restrict pCond,    /* condition variable  */
    pthread_mutex_t * _Restrict pMutex,   /* POSIX thread mutex  */
    const struct timespec * _Restrict 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 it is not then this function returns an error (**EPERM**).

If the condition variable is signaled 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 signaled, 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). However the mutex will 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 one of the following non-zero error code: **EPERM**, **EINVAL**, **ETIMEDOUT**

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_wait( )**

pthread_cond_wait( )

NAME **pthread_cond_wait( )** – wait for a condition variable (POSIX)

SYNOPSIS

```
int pthread_cond_wait
    (
    pthread_cond_t * _Restrict pCond,  /* condition variable   */
    pthread_mutex_t * _Restrict pMutex  /* POSIX thread mutex   */
    )
```

DESCRIPTION This function atomically releases the mutex *pMutex* and waits for the condition variable *pCond* to be signaled 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 (**EPERM**).

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). However the mutex will be released prior to the thread exiting so that this mutex can be used by other threads.

RETURNS On success zero; on failure the **EPERM** or **EINVAL** error codes.

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 On success zero; on failure the **EINVAL** error code.

ERRNO
N/A

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 the **EINVAL** error code.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_cond_init( )**, **pthread_condattr_destroy( )**

pthread_create( )

NAME
pthread_create( ) – create a thread (POSIX)

SYNOPSIS
```
int pthread_create
    (
    pthread_t *       restrict pThread,  /* Thread ID (out) */
    const pthread_attr_t * _Restrict pAttr,    /* Thread attributes object */
    void * (*startRoutine)(void *),       /* Entry function */
    void * _Restrict          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, or when the scheduling attributes of the underlying VxWorks task for either this thread or its parent cannot be set.

ESRCH
the underlying VxWorks task for this thread has died or been removed before the thread creation operation was finished.

ENOSYS
the thread creation cannot be achieved because the POSIX scheduler has not been configured in the system.

ERRNO
Not Available

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 one of the following non-zero error codes: **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.

If the calling task is not a POSIX thread, it will exit immediately.

RETURNS

Does not return.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_cleanup_push( )**, **pthread_detach( )**, **pthread_join( )**, **pthread_key_create( )**

pthread_getconcurrency( )

NAME
pthread_getconcurrency( ) – get the level of concurrency (POSIX)

SYNOPSIS
```
int pthread_getconcurrency (void)
```

DESCRIPTION
This routine retrieves the concurrency level as set by the previous call to **pthread_setconcurrency( )** function.

NOTE
VxWorks does not support multi-level scheduling; the **pthread_setconcurrency( )** and **pthread_getconcurrency( )** functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the level parameter is saved when **pthread_setconcurrency( )** is called so that a subsequent call to **pthread_getconcurrency( )** shall return the same value.

RETURNS
the value set by a previous call to the **pthread_setconcurrency( )** function. If the **pthread_setconcurrency( )** function was not previously called, this function will return zero.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_setconcurrency( )**

pthread_getschedparam( )

NAME
pthread_getschedparam( ) – get value of schedparam attribute from a thread (POSIX)

SYNOPSIS
```
int pthread_getschedparam
    (
    pthread_t                 thread,   /* thread                 */
    int * _Restrict           pPolicy,  /* current policy (out) */
    struct sched_param * _Restrict 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*.

RETURNS
On success zero; on failure the **ESRCH** or **ENOSYS** error codes.

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.

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 one of the following non-zero error codes: **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 **taskKill( )** 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 thread 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 one of the following non-zero error codes: **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
```

```
    (
    const pthread_mutex_t * _Restrict pMutex,       /* POSIX mutex
*/
    int * _Restrict           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 * _Restrict pMutex,  /* pthread mutex */
    const pthread_mutexattr_t * _Restrict 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.

NOTE
this routine does not verify whether the *pMutex* parameter corresponds to an already initialized mutex object. It is up to the caller to ensure that *pMutex* does not correspond to an object already in use.

RETURNS
On success zero; on failure one of the following non-zero error codes: **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 a thread attempts to relock a mutex that it has already locked and - if the mutex type is **PTHREAD_MUTEX_NORMAL**, pthread_mutex_lock will deadlock on itself and the thread will block indefinitely. - if the mutex type is **PTHREAD_MUTEX_ERRORCHECK**, pthread_mutex_lock will return **EDEADLK** error. - if the mutex type is **PTHREAD_MUTEX_RECURSIVE**, pthread_mutex_lock will increment its **lock count** and return success.

The mutex type **PTHREAD_MUTEX_DEFAULT** is mapped to **PTHREAD_MUTEX_NORMAL**.

RETURNS On success zero; on failure one of the following non-zero error codes: **EINVAL**, **EDEADLK**

ERRNO N/A

SEE ALSO **pthreadLib**, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_timedlock( )**, **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 * _Restrict pMutex,         /* POSIX mutex
*/
    int                       prioceiling,     /* new priority ceiling
*/
    int * _Restrict           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 one of the following non-zero error codes: **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_timedlock( )

NAME

pthread_mutex_timedlock( ) – lock a mutex with timeout (POSIX)

SYNOPSIS

```
int pthread_mutex_timedlock
    (
    pthread_mutex_t * _Restrict pMutex,   /* POSIX mutex          */
    const struct timespec * _Restrict pAbstime  /* timeout time         */
    )
```

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_timedlock( )** 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 or the specified timeout expires, whichever occurs earlier. The timeout is specified by *pAbstime* parameter. In the case of timeout expiration, the thread is unblocked and **ETIMEDOUT** returned.

If a thread attempts to relock a mutex that it has already locked and - if the mutex type is **PTHREAD_MUTEX_NORMAL**, pthread_mutex_timedlock will deadlock on itself and the thread will block indefinitely. - if the mutex type is **PTHREAD_MUTEX_ERRORCHECK**, pthread_mutex_timedlock will return the **EDEADLK** error. - if the mutex type is

PTHREAD_MUTEX_RECURSIVE, pthread_mutex_timedlock will increment its **lock count** and return success.

The mutex type **PTHREAD_MUTEX_DEFAULT** is mapped to **PTHREAD_MUTEX_NORMAL**.

RETURNS On success zero; on failure one of the following non-zero error code: **EINVAL**, **ETIMEDOUT**, **EDEADLK**

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_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**.

If a thread attempts to relock a mutex that it has already locked and - if the mutex type is **PTHREAD_MUTEX_NORMAL** or **PTHREAD_MUTEX_ERRORCHECK**, pthread_mutex_trylock returns immediately with the error code **EBUSY**. - if the mutex type is **PTHREAD_MUTEX_RECURSIVE**, pthread_mutex_trylock will increment its **lock count** and return success.

The mutex type **PTHREAD_MUTEX_DEFAULT** is mapped to **PTHREAD_MUTEX_NORMAL**.

RETURNS On success zero; on failure one of the following non-zero error codes: **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 one of the following non-zero error codes: **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
    (
    const pthread_mutexattr_t * _Restrict pAttr,        /* mutex attributes
*/
    int * _Restrict           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
    (
    const pthread_mutexattr_t * _Restrict pAttr,     /* mutex attributes */
    int * _Restrict           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_gettype( )

NAME

pthread_mutexattr_gettype( ) – get the current value of the type attribute in a mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_gettype
    (
    const pthread_mutexattr_t * _Restrict pAttr,  /* mutex attributes    */
    int * _Restrict           pType   /* current mutex type (out) */
    )
```

DESCRIPTION

This function gets the current value of the type 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_init( )**, **pthread_mutexattr_settype( )**, **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 as defined by the POSIX specification:

Mutex Protocol

PTHREAD_PRIO_NONE - priority and scheduling of the owner thread are not affected by its mutex ownership.

Mutex Type
PTHREAD_MUTEX_DEFAULT - no deadlock detection.

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 one of the folowing non-zero error codes: **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_mutexattr_settype( )

NAME

pthread_mutexattr_settype( ) – set type attribute in mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_settype
    (
    pthread_mutexattr_t *pAttr,  /* mutex attributes     */
    int                 type     /* mutex type           */
    )
```

DESCRIPTION

This function sets the type attribute in a mutex attributes object. The default value of the type attribute is **PTHREAD_MUTEX_DEFAULT**. Valid mutex types are:

PTHREAD_MUTEX_NORMAL
deadlock detection is not provided; attempt to relock causes deadlock; attempt to unlock a mutex owned by another thread or unlock a unlocked mutex returns error.

PTHREAD_MUTEX_ERRORCHECK
error checking is provided; attempt to relock a mutex or unlock a mutex owned by another thread or unlock a unlocked mutex returns error.

PTHREAD_MUTEX_RECURSIVE
can be relocked by a thread.

PTHREAD_MUTEX_DEFAULT
set to **PTHREAD_MUTEX_NORMAL** in VxWorks implementation.

RETURNS

On success zero; on failure the **EINVAL** error code.

ERRNO
N/A

SEE ALSO
pthreadLib, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_gettype( )**, **pthread_mutex_lock( )**, **pthread_mutex_timedlock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **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. This is not true however if the initialization routine is a cancellation point and has been cancelled: in that case *pOnceControl* will be left as if **pthread_once( )** was never called (see above).

Also there is no guarantee that the thread executing the initialization routine is the first one to return from **pthread_once( )** in case of concurrent calls by multiple threads for the same once control variable.

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 zero on success, **EINVAL** otherwise.

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_setconcurrency( )

NAME **pthread_setconcurrency( )** – set the level of concurrency (POSIX)

SYNOPSIS

```
int pthread_setconcurrency
```

```
    (
    int level
    )
```

DESCRIPTION This routine changes the concurrency level as described by the *level* argument. If *level* is negative, a **EINVAL** error code is returned.

NOTE VxWorks does not support multi-level scheduling; the **pthread_setconcurrency( )** and **pthread_getconcurrency( )** functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the level parameter is saved when **pthread_setconcurrency( )** is called so that a subsequent call to **pthread_getconcurrency( )** shall return the same value.

RETURNS On success zero; on failure a **EINVAL** error code is returned.

ERRNO N/A

SEE ALSO **pthreadLib**, **pthread_getconcurrency( )**

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*.

This implementation does not support dynamically changing the thread's scheduling policy to **SCHED_SPORADIC** and will return **ENOTSUP** if an attempt is made.

This implementation return **EPERM** in the case of an attempt to change the thread's priority while it is holding a mutex using the priority ceiling protocol.

RETURNS On success zero; on failure one of the following non-zero error codes: **EINVAL**, **ESRCH**, **ENOTSUP** or **EPERM**.

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_setschedprio( )

NAME **pthread_setschedprio( )** – dynamically set priority attribute for a thread (POSIX)

SYNOPSIS

```
int pthread_setschedprio
    (
    pthread_t thread,   /* thread               */
    int       prio      /* new priority         */
    )
```

DESCRIPTION This routine will set the priority *prio* for the thread specified by *thread*

This implementation return **EPERM** in the case of an attempt to change the thread's priority while it is holding a mutex using the priority ceiling protocol.

RETURNS On success zero; on failure one of the following non-zero error codes: **EINVAL**, **ESRCH** or **EPERM**

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 one of the following 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 * _Restrict set,  /* new set of signals            */
    sigset_t * _Restrict       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( )**

---

putenv( )

NAME

putenv( ) – change or add a value to the environment

SYNOPSIS

```
int putenv
    (
    char * pEnvString  /* Environment variable name */
    )
```

DESCRIPTION

This routine adds a new environment variable and value to the global environment if the variable does not already exist. If the variable already exists, it updates the value. The string argument should be in the form "name=value". It also excepts spaces in the string, " name = value".

Unlike the POSIX implementation, the string passed as a parameter is copied to a private buffer.

RETURNS

0 for success, **ENOMEM** if the memory cannot be allocated for the string, -1 if environment variable name is **NULL** or if there isn't a value assigned

ERRNO

N/A

SEE ALSO

setenv, **setenv( )**, **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*.

pxClose( )

NAME

pxClose( ) – close a reference to a POSIX semaphore or message queue (syscall)

SYNOPSIS

```
int pxClose
    (
    OBJ_HANDLE  handle
    )
```

DESCRIPTION

This routine closes the specified *handle* to the underlying POSIX object. If the handle refers to an unnamed semaphore, then the object is deleted, provided no task is blocked on it. If tasks are blocked on the semaphore then this function returns **ERROR** with **EBUSY** errno.

RETURNS

OK or **ERROR** if unsuccessful.

ERRNO

EINVAL
Invalid *handle* specified.

EBUSY
Attempt to delete an unnamed semaphore but tasks are blocked on it.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO **posixScLib**, **pxOpen( )**, **pxUnlink( )**

pxCtl( )

NAME **pxCtl( )** – control operations on POSIX semaphores and message queues (syscall)

SYNOPSIS

```
STATUS pxCtl
    (
    OBJ_HANDLE      handle,
    PX_CTL_CMD_CODE cmdCode,
    void *          pArgs,
    UINT *          pArgSize
    )
```

DESCRIPTION This routine performs various operations on POSIX objects, as specified by the *cmdCode*. The object is specified by its *handle*. The arguments are passed in a buffer *pArgs* whose length is specified in *pArgSize*.

The control operation is specified using *cmdCode*. The possible values are:

PX_MQ_NOTIFY
This operation is the helper function for **mq_notify( )**. The argument buffer *pArgs* contains the following items.

```
const struct sigevent * pNotification;
```

PX_SEM_GETVALUE
This operation is the helper function for **sem_getvalue( )**. The argument buffer *pArgs* contains the following items.

```
int * sval;
```

PX_MQ_ATTR_GET
This operation is the helper function for **mq_getattr( )** and **mq_setattr( )**. It gets the message queue attributes of message size, maximum messages allowed on this queue and the number of messages currently in the queue. The argument buffer *pArgs* contains the following items.

```
struct mq_attr * pOldMqStat;
```

RETURNS **OK** or **ERROR** if unsuccessful

ERRNO

EBADF
Invalid *handle* specified for message queue operations.

EINVAL
Invalid *handle* specified for semaphore operations.

EBUSY
PX_MQ_NOTIFY is specified and a task is already registered for notification by the specified message queue.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO
posixScLib, **mq_getattr( )**, **mq_setattr( )**, **mq_notify( )**, **sem_getvalue( )**

pxMqReceive( )

NAME
pxMqReceive( ) – receive a message from a POSIX message queue (syscall)

SYNOPSIS
```
ssize_t pxMqReceive
    (
    OBJ_HANDLE        handle,
    char   *          pMsg,
    size_t            msgLen,
    unsigned int *    pMsgPrio,
    PX_WAIT_OPTION    waitOption,
    struct timespec * timeOut
    )
```

DESCRIPTION
This routine receives a message from the POSIX message queue referred by *handle*. The message is copied into the buffer *pMsg*. The length of the buffer is specified in *msgLen*, which should not be less than the message size attribute of the message queue. The priority of the received message is returned in *pMsgPrio*. The timeout is specified using *waitOption* and the *timeOut*. The *waitOption* can be **PX_NO_WAIT** or **PX_WAIT_FOREVER**. Timed waiting is not supported. The *timeOut* parameter must be set to **NULL**.

RETURNS
OK or **ERROR** if unsuccessful.

ERRNO
ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

EBADF
Invalid *handle* specified.

EMSGSIZE
The specified message size *msgLen* is less than the message size attribute of the message queue.

EAGAIN
PX_NO_WAIT is specified in *waitOption* and the message queue is empty.

SEE ALSO **posixScLib**, **pxMqSend( )**, **pxOpen( )**, **mq_receive( )**

pxMqSend( )

NAME **pxMqSend( )** – send a message to a POSIX message queue (syscall)

SYNOPSIS

```
int pxMqSend
    (
    OBJ_HANDLE        handle,
    const char *      pMsg,
    size_t            msgLen,
    unsigned int      msgPrio,
    PX_WAIT_OPTION    waitOption,
    struct timespec * timeOut
    )
```

DESCRIPTION This routine sends a message to the POSIX message queue referred by *handle* from the buffer *pMsg*. The length of the buffer is specified in *msgLen* and it should not be greater than the message size attribute of the message queue. The message is sent with a priority of *msgPrio*. The timeout is specified using *waitOption* and the *timeOut*. The *waitOption* can be **PX_NO_WAIT** or **PX_WAIT_FOREVER**. Timed waiting is not supported. The *timeOut* parameter must be set to **NULL**.

RETURNS **OK** or **ERROR** if unsuccessful.

ERRNO

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

EBADF
Invalid *handle* specified.

EAGAIN
PX_NO_WAIT is specified in *waitOption* and the message queue is full.

EMSGSIZE
The specified message size *msgLen* is more than the message size attribute of the message queue.

EINVAL
The value of *msgPrio* is outside the valid range.

SEE ALSO **posixScLib**, **pxMqReceive( )**, **pxOpen( )**, **mq_send( )**

pxOpen( )

NAME

pxOpen( ) – open a POSIX semaphore or message queue (syscall)

SYNOPSIS

```
OBJ_HANDLE pxOpen
    (
    PX_OBJ_TYPE  type,
    const char * name,
    int          mode,
    void *       attr
    )
```

DESCRIPTION

This routine opens a POSIX object depending upon the specified *type*, which can be one of the following.

PX_MQ
POSIX message queue

PX_SEM
POSIX semaphore

It searches the name space and returns a handle to an existing object 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 **pxOpen( )** can perform the search, and which are selected depending on the state of the **OM_PUBLIC** flag: the "private to the application" name space, and the "public" name space in which public objects can be found. If no name is specified for the object then it must be in the private name space. Setting the **OM_PUBLIC** flag for such an object results in error.

The *mode* parameter passed to this routine consists of the access rights (currently not implemented), and the opening flags, which are a bitwise-inclusive-OR of the following.

OM_CREATE
Creates the object if none is found.

OM_EXCL
When set jointly with the **OM_CREATE** flag, creates a new object without trying to open an existing one. The call fails if the object's name causes a name clash. This flag has no effect if the flag **OM_CREATE** is not set.

OM_PUBLIC
A public object with the same name is created or searched for. If this flag is not set, an object private to the calling application with the same name is created or searched for.

OM_RECLAIM_DISABLE
The created object will not participate in the automatic resource reclamation mechanism. This flag has no effect if the **OM_PUBLIC** flag is not set.

The *attr* parameter contains different values depending upon the *type*.

PX_MQ
attr is a pointer to struct mq_attr as defined in **mqueue.h**.

PX_SEM
attr is an integer containing the initial value of the semaphore.

RETURNS A handle to the opened object if successful, or **ERROR** if unsuccessful.

ERRNO

EEXIST
OM_CREATE and **OM_EXCL** specified but the name already exists.

ENOENT
OM_CREATE not specified and the name does not exist.

ENOSPC
Failure due to resource constraints.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO **posixScLib**, **pxClose( )**, **pxUnlink( )**, **mq_open( )**, **sem_open( )**

pxSemPost( )

NAME **pxSemPost( )** – post a POSIX semaphore (syscall)

SYNOPSIS

```
int pxSemPost
    (
    OBJ_HANDLE handle
    )
```

DESCRIPTION This routine posts the POSIX semaphore specified by *handle*. The semaphore can be named or unnamed.

RETURNS **OK** or **ERROR** if unsuccessful.

ERRNO

EINVAL
Invalid semaphore *handle* specified.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO **posixScLib**, **pxSemWait( )**, **pxOpen( )**, **sem_post( )**

pxSemWait( )

NAME

pxSemWait( ) – wait for a POSIX semaphore (syscall)

SYNOPSIS

```
int pxSemWait
    (
    OBJ_HANDLE        handle,
    PX_WAIT_OPTION    waitOption,
    struct timespec * timeOut
    )
```

DESCRIPTION

This routine obtains the POSIX semaphore specified by *handle*. If the semaphore is not available, the calling task is blocked for period specified by *waitOption* and *timeOut*. The *waitOption* can be **PX_NO_WAIT** or **PX_WAIT_FOREVER**. Timed wait is not supported. The *timeOut* parameter must be set to **NULL**.

RETURNS

OK or **ERROR** if unsuccessful.

ERRNO

EINVAL
Invalid semaphore *handle* specified.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO

posixScLib, **pxSemPost( )**, pxOpen, **sem_wait( )**

pxUnlink( )

NAME

pxUnlink( ) – unlink the name of a POSIX semaphore or message queue (syscall)

SYNOPSIS

```
int pxUnlink
    (
    PX_OBJ_TYPE  type,
    const char * name
    )
```

DESCRIPTION

This routine removes the specified *name* from the object name space for the specified *type* of POSIX object. Subsequent attempts to open a reference to an object with this name will result in the opening of a different object, subject to the rules specified in **pxOpen( )**.

RETURNS

OK or **ERROR** if unsuccessful.

ERRNO

ENOENT
An object with the specified *name* of the specified *type* does not exist.

ENOSYS
The **INCLUDE_POSIX_SEM** and **INCLUDE_POSIX_MQ** components have not been configured into the system.

SEE ALSO

posixScLib, **pxOpen( )**, **pxClose( )**

raise( )

NAME

raise( ) – send a signal to the calling RTP (POSIX)

SYNOPSIS

```
int raise
    (
    int signo  /* signal to send to caller's RTP */
    )
```

DESCRIPTION

This routine sends the signal *signo* to the calling RTP. Any task in the target RTP that has unblocked *signo* can receive the signal.

RETURNS

0, or -1 if the signal number is invalid.

ERRNO

EINVAL
The value of the *sig* argument is an invalid signal number.

SEE ALSO

sigLib, **taskRaise( )**, **rtpRaise( )**

read( )

NAME

read( ) – read bytes from a file or device

SYNOPSIS

```
ssize_t read
    (
    int    fd,
    void * buffer,
    size_t maxbytes
    )
```

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 from the RTP heap (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.

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:

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

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

remove( )

NAME **remove( )** – remove a file (ANSI) (syscall)

SYNOPSIS

```
int remove
   (
   const char * name
   )
```

DESCRIPTION This routine deletes a specified file. It calls the driver for the particular device on which the file is located to do the work.

RETURNS **OK** if there is no delete routine for the device or the driver returns **OK**; **ERROR** if there is no such device or the driver returns **ERROR**.

ERRNO Not Available

SEE ALSO **ioLib**, *"American National Standard for Information Systems -"*, *"Programming Language - C, ANSI X3.159-1989: Input/Output (***stdio.h***),* "

---

rename( )

NAME **rename( )** – change the name of a file

SYNOPSIS

```
int rename
    (
     const char *oldname,   /* path name of the file to be renamed  */
     const char *newname    /* path name to which to rename the 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 supports **rename( )**, but **netDrv** and **nfsDrv** do 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**

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**

rtpExit( )

NAME **rtpExit( )** – terminate the calling process

SYNOPSIS

```
void rtpExit
    (
    int status
    )
```

DESCRIPTION This routine terminates the calling RTP. This function is currently aliased to **exit( )**, and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

RETURNS N/A.

ERRNO

SEE ALSO **rtpLib**, **exit( )**, **_exit( )**, **_Exit( )**, the VxWorks programmer guides

rtpInfoGet( )

NAME **rtpInfoGet( )** – Get specific information on an RTP (syscall)

SYNOPSIS

```
RTP_ID rtpInfoGet
   (
   RTP_ID rtpId,                  /* RTP_ID to get info for */
```

```
   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 *rtpId* parameter may be left null to get information about the current RTP.

The information stored in the descriptor, for the most part, is a 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 tasks are private to an RTP.

The IDs of the initTaskId and parentId are the opaque IDs in user space. To display information on these IDs from the shell, use **objHandleShow( )**.

RETURNS

OK or **ERROR**

ERRNOS

S_objLib_OBJ_ID_ERROR
   Invalid RTP ID or null *rtpStruct* parameter.

SEE ALSO

rtpLib, **rtpShow( )**

rtpIoTableSizeGet( )

NAME

rtpIoTableSizeGet( ) – get *fd* table size for given RTP

SYNOPSIS

```
size_t rtpIoTableSizeGet
    (
    RTP_ID rtpId
    )
```

DESCRIPTION This routine returns the size of the *fd* table for the specified RTP. A value of 0 for *rtpId* implies the currently running RTP.

RETURNS number of *fd* table entries in specified RTP.

ERRNO N/A

SEE ALSO **ioLib**

rtpIoTableSizeSet( )

NAME **rtpIoTableSizeSet( )** – set *fd* table size for given RTP

SYNOPSIS

```
STATUS rtpIoTableSizeSet
    (
    RTP_ID rtpId,
    size_t newSize
    )
```

DESCRIPTION This routine can be used to enlarge the FD table if necessary. If the requested size for the table is larger than the current size, then a block of memory is allocated for the table. If the requested size is smaller, no reallocation is performed. The actual table size is accessible using **rtpIoTableSizeGet( )**.

RETURNS Returns **OK**, or **ERROR** if requested size is less than 3, or if new memory could not be allocated.

ERRNO N/A

SEE ALSO **ioLib**

rtpKill( )

NAME **rtpKill( )** – send a signal to a RTP

SYNOPSIS

```
int rtpKill
  (
  RTP_ID  rtpId,
  int signo
  )
```

DESCRIPTION This routine sends a signal *signo* to the RTP specified by *rtpId*. Any task in the target RTP that has unblocked *signo* can receive the signal. 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 RTP ID or signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **kill( )**, **taskKill( )**

rtpRaise( )

NAME **rtpRaise( )** – send a signal to the calling RTP

SYNOPSIS

```
int rtpRaise
    (
    int signo    /* signal to send to caller's RTP */
    )
```

DESCRIPTION This routine sends the signal *signo* to the RTP. Any task in the target RTP that has unblocked *signo* can receive the signal. This routine is currently aliased to **raise( )** and is provided as a convenience to achieve uniform meaning across both kernel and user-mode code.

RETURNS 0, or -1 if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **taskRaise( )**, **rtpRaise( )**

rtpSigqueue( )

NAME **rtpSigqueue( )** – send a queued signal to a RTP

SYNOPSIS

```
int rtpSigqueue
    (
    int rtpId,
    int signo,
    const union sigval value
    )
```

DESCRIPTION

This routine sends the signal *signo* with the signal-parameter value *value* to the process *rtpId*. Any task in the target RTP that has unblocked *signo* can receive the signal. 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 RTP ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO

EINVAL
EAGAIN

SEE ALSO

sigLib, **sigqueue( )**

---

rtpSpawn( )

NAME

rtpSpawn( ) – spawns a new Real Time Process (RTP) in the system (syscall)

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 RTPs 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 an 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, 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.

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.

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.

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**

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**, **SCHED_RR**, **SCHED_SPORADIC** or **SCHED_OTHER**).

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**, **SCHED_RR**, **SCHED_SPORADIC** or **SCHED_OTHER**).

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.

CAVEAT this routine does not support the POSIX thread scheduler. Pthreads should use the **pthread_getschedparam( )** API instead.

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**).

CAVEAT this routine does not support the POSIX thread scheduler. Pthreads should use the **pthread_getschedparam( )** API instead.

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 scheduler's current time slice period. This time information may be 0 second and 0 nanosecond when the native VxWorks scheduler is used, by opposition to the POSIX thread scheduler, and it is not set in round-robin mode.

When the *tid* parameter is set to null, the caller's ID is automatically used.

RETURNS

0 (**OK**) if successful, -1 (**ERROR**) on error.

ERRNO

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.

CAVEAT this routine does not support the POSIX thread scheduler. Pthreads should use the **pthread_setschedparam( )** API instead.

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.

CAVEAT this routine does not support the POSIX thread scheduler. Pthreads should use the **pthread_setschedparam( )** API instead.

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 N/A

SEE ALSO **schedPxLib**

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_t      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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute
SD_ATTR_RX	Read/Execute
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.

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_CACHE_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.

ERRNOS 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_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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO **sdLib**, **sdOpen( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

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.

ERRNOS 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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO **sdLib**, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, **sdProtect( )**

sdInfoGet( )

NAME **sdInfoGet( )** – Get specific information about a shared data region

SYNOPSIS

```
STATUS sdInfoGet
    (
    SD_ID     sdId,             /* ID of shared data region */
    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.

ERRNOS

Possible errno values set by this routine are:

S_sdLib_INVALID_SD_ID
sdId is not valid

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

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.

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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute
SD_ATTR_RX	Read/Execute
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.

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_CACHE_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.

ERRNOS

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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

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_t      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, excepting *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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute
SD_ATTR_RX	Read/Execute
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.

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_CACHE_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.

ERRNOS

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_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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO **sdLib**, **sdCreate( )**, **sdUnmap( )**, **sdProtect( )**, **sdDelete( )**

sdProtect( )

NAME **sdProtect( )** – Change the protection attributes of a mapped shared data region

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
SD_ATTR_RO	Read Only
SD_ATTR_RWX	Read/Write/Execute
SD_ATTR_RX	Read/Execute
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.

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_CACHE_MSK

MMU_ATTR_SPL_MSK

Care must be taken to provide suitable values for all these attributes.

RETURNS

OK, or **ERROR** on failure.

ERRNOS

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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO

sdLib, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdUnmap( )**, **sdDelete( )**

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. There are currently no options specified for this function, zero should be passed in the options parameter.

RETURNS

OK, or **ERROR** on failure.

ERRNOS

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

ENOSYS
INCLUDE_SHARED_DATA has not been configured into the kernel.

SEE ALSO

sdLib, **sdCreate( )**, **sdOpen( )**, **sdMap( )**, **sdProtect( )**, **sdDelete( )**

select( )

NAME

select( ) – pend on a set of file descriptors (syscall)

SYNOPSIS

```
int select
   (
   int               width,
   fd_set *          pReadFds,
   fd_set *          pWriteFds,
   fd_set *          pExceptFds,
   struct timeval * pTimeOut
   )
```

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 *pExceptFds* parameter is currently unused, but is provided for UNIX call compatibility.)

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* or *pWriteFds* 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. Also, **select( )** now examines write file descriptors in addition to read file descriptors; however, exception file descriptors remain unsupported.

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.

SEE ALSO
ioLib, the VxWorks programmer guides

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).

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_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 a blocked task on a semaphore created with this option would wakeup the task. The returns then returns **ERROR** with errno set to **EINTR**. This option is off by default.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated.

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.

SEE ALSO

semLib, **taskSafe( )**, **taskUnsafe( )**

---

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 initial count specified by *initialCount*.

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_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 a blocked task on a semaphore created with this option would wakeup the task. The returns then returns **ERROR** with errno set to **EINTR**. This option is off by default.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated.

ERRNO

S_semLib_INVALID_OPTION
: Invalid option was specified.

S_semLib_INVALID_INITIAL_COUNT
: The specified initial count is negative

S_memLib_NOT_ENOUGH_MEMORY
: Not enough memory available to create the semaphore.

S_semLib_INVALID_QUEUE_TYPE
: Invalid type of semaphore queue specified.

SEE ALSO

semLib, **taskSafe( )**, **taskUnsafe( )**

semClose( )

NAME

semClose( ) – close a named semaphore

SYNOPSIS

```
STATUS semClose
    (
    SEM_ID semId  /* semaphore ID to delete */
    )
```

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_CALL** option.

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.

SEE ALSO

semLib, **semOpen( )**, **semUnlink( )**

---

semCtl( )

NAME

semCtl( ) – perform a control operation against a kernel semaphore (system call)

SYNOPSIS

```
STATUS semCtl
    (
    SEM_ID          semId,      /* kernel semaphore id */
    VX_SEM_CTL_CMD  command,    /* command to run */
    void *          pArg,       /* pointer to argument */
    UINT *          pArgSize    /* pointer to argument size */
    )
```

DESCRIPTION

The **semCtl( )** system call performs the requested *command* against the kernel semaphore specified by *semId*. The following is a description of the supported commands:

VX_SEM_CTL_MTAKE_PROXY
Takes the empty kernel mutex semaphore specified by *semId* on behalf of the task identified by *pArg*. Both the semaphore and task id provided must be local to the current RTP context. The argument *pArgSize* is not used for this command.

VX_SEM_CTL_SEM_OWNER
Returns in *pArg* the task id of the current owner of the specified kernel semaphore, or **NULL** if the semaphore is currently unowned. The semaphore must be a private semaphore which is local to the RTP. The argument *pArgSize* is not used for this

command. The space pointed to by *pArg* must be large enough to hold the value of a task id.

VX_SEM_CTL_FLUSH
Atomically unblocks all tasks pended on the specified kernel semaphore; the state of the underlying kernel semaphore is unchanged. All pended tasks will enter the ready queue before having a chance to execute. The arguments *pArg* and *pArgSize* are not used for this command. This command cannot be issued against a kernel semaphore which is not local to the calling RTP.

WARNING
The semaphore id which is used must be the id returned from the **_semOpen( )** system call. The semaphore ids in the kernel space are distinct from the values returned by **_semOpen( )** and cannot be used with this system call.

RETURNS
OK if the requested operation completes successfully, otherwise **ERROR**.

RETURNS
OK, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNO
S_objLib_OBJ_ID_ERROR
The *semId* parameter is an invalid semaphore ID or *pArg* is invalid for the requested operation.

S_objLib_OBJ_INVALID_ARGUMENT
In commands in which *pArg* is needed, like **VX_SEM_CTL_SEM_OWNER**, the buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden; Or it does belong to this RTP task but the needed accesses, read, write or both, are not allowed due to access control. In **VX_SEM_CTL_SEM_OWNER**, write is needed.

S_semLib_INVALID_OPERATION
The requested operation is not valid.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to perform the requested control command.

SEE ALSO
semLib, **_semOpen( )**, **semFlush( )**

---

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 the 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**.

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** if success, **ERROR** otherwise.

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_OPERATION_UNSUPPORTED
Deleting a named semaphore is not permitted.

SEE ALSO **semLib**

---

semEvStart( )

NAME **semEvStart( )** – start 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 *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)
Allows subsequent registrations from other tasks to 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
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 semDelete to be performed between the calls to semEvStart and eventReceive. This will prevent 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
Routine has been called from interrupt level.

S_eventLib_EVENTSEND_FAILED
User chooses to send events right away and that operation is failed. **OK** may be returned if registration is successful.

S_eventLib_ZERO_EVENTS
User passed in a value of zero to the *events* parameter.

SEE ALSO
semEvLib, **eventLib**, **semLib**, **semEvStop( )**

---

semEvStop( )

NAME
semEvStop( ) – stop 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 has to be called from the task that is already registered on that particular semaphore.

RETURNS

OK on success, or **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
The semaphore ID is invalid.

S_intLib_NOT_ISR_CALLABLE
Routine has been called at interrupt level.

S_eventLib_TASK_NOT_REGISTERED
Routine has not been 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.

Currently only binary and mutex semaphore types are supported by **semExchange( )**.

User level semaphores are not yet supported. An attempt to exchange a user level sempaphore will result a return value of **ERROR**.

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.

RETURNS

OK, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNOS

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_OPTION
Invalid option 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

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.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the operation is not supported.

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_semLib_INVALID_OPERATION
Operation not supported on semaphore type.

SEE ALSO **semLib**, **semTake( )**, **semGive( )**, the VxWorks programmer guides.

semGive( )

NAME **semGive( )** – give a semaphore

SYNOPSIS

```
STATUS semGive
    (
    SEM_ID semId  /* semaphore ID to give */
    )
```

DESCRIPTION This routine performs the give operation on the 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 sempahore 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.

RETURNS **OK** on success or **ERROR** otherwise

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_semLib_INVALID_OPERATION
Current task not owner of mutex 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.

S_semLib_INVALID_OPTION
Semaphore type is invalid

SEE ALSO **semLib**, **_semTake( )**, **_semGive( )**, **semTake( )**, **semBLib**, **semCLib**, **semMLib**, the VxWorks programmer guides.

---

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 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 (couting sems) */
        BOOL full;          /* OUT: binary semaphore FULL? */
        int  owner;         /* OUT: mutex semaphore owner */
```

```
        } state;
    } 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 task ID of the task that owns the mutex *state.owner* only if that task resides in the same RTP as the calling task. If the owner task resides in another RTP, *state.owner* is set to -1. If there is no owner, it is set to 0.

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 owned, then there may be tasks blocked on **semTake( )**. The *numTasks* field indicates the number of blocked tasks.

The *options* field is the parameter with which the semaphore was created.

Obtaining a list of the task IDs of tasks blocked on the semaphore is not supported from user space which is why the **SEM_INFO** structure definition shown above is different than the one used in kernel space.

WARNING The information returned by this routine is not static and may be obsolete by the time it is examined. 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.

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

ERRNO **S_objLib_OBJ_ID_ERROR**
Invalid semaphore ID.

SEE ALSO **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 a blocked task on a semaphore created with this option would wakeup the task. The returns then returns **ERROR** with errno set to **EINTR**. This option is off by default.

SEM_KERNEL (0x100)
semTake( ) and **semGive( )** operations will operate directly on the kernel side semaphore. This results in a system call always being issued for taking and giving a semaphore, regardless of the state of the semaphore.

SEM_USER (0x200)
semTake( ) and **semGive( )** operations for an uncontested semaphore will be performed in user space rather than issue a system call to handle the operation. This significantly improves the performance of taking and giving a semaphore when there are no other tasks blocked on the semaphore. Contested semaphores are always handled via a system call into the kernel.

SMP CONSIDERATIONS

For SMP, **SEM_USER** option has no effect. All semaphores created for SMP will default to type **SEM_KERNEL**.

RETURNS The semaphore ID, or **NULL** if the semaphore cannot be created.

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_QUEUE_TYPE
Invalid type of semaphore queue specified.

SEE ALSO **semLib**, **taskSafe( )**, **taskUnsafe( )**

semOpen( )

NAME **semOpen( )** – open a named semaphore

SYNOPSIS

```
SEM_ID semOpen
    (
    const char * name,       /* semaphore name */
    SEM_TYPE     type,       /* semaphore type - mutex, binary, counting */
    int          initState,  /* state after creation */
    int          options,    /* creation options */
    int          mode,       /* creation mode */
    void *       context     /* context value */
    )
```

DESCRIPTION

This function 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
SEM_TYPE_RW	create a read/write 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*. For read/write semaphores the maximum number of readers is set to *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.

LIMITATIONS All semaphores created by this routine are kernel level semaphores. Therefore all operations (e.g. semTake/semGive) incur in a system call.

RETURNS The **SEM_ID** of the opened semaphore, or **NULL** if unsuccessful.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP to open the semaphore.

S_semLib_INVALID_OPTION
Invalid option was passed for semaphore creation.

S_semLib_INVALID_STATE
Invalid initial state for binary semaphore creation.

S_semLib_INVALID_INITIAL_COUNT
The specified initial count for counting semaphore is negative.

S_semLib_INVALID_QUEUE_TYPE
Invalid type of semaphore queue specified.

S_semLib_INVALID_OPERATION
Invalid type of semaphore requested.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the semaphore handle.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid. *name* buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

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.

SEE ALSO

semLib, **semUnlink( )**, **semClose( )**, **semTake( )**, **semGive( )**, **semCtl( )**, the VxWorks programmer guides.

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.

WARNING

This routine may 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,     /* RW-semaphore options */
    int maxReaders   /* Max number of readers for semaphore */
    )
```

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( )**

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 the 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**).

A task pended on a semaphore created with **SEM_INTERRUPTIBLE** option receives a signal, returns **ERROR** with errno set to **EINTR**.

RETURNS

OK, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNO

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_objLib_OBJ_UNAVAILABLE,
Would have blocked but **NO_WAIT** was specified.

S_objLib_OBJ_TIMEOUT
Timeout occured while pending on sempahore.

S_semLib_INVALID_OPTION
Semaphore type is invalid

EINTR
Signal received while blocking on the semaphore

SEE ALSO

semLib, **_semTake( )**, **_semGive( )**, **semGive( )**, **semBLib**, **semCLib**, **semMLib**, the VxWorks programmer guides.

semUnlink( )

NAME **semUnlink( )** – unlink a kernel 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.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO

S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL**. *name* buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

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.

SEE ALSO **semLib**, **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.

WARNING This routine may 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.

A given semaphore can be opened multiple times by calling **sem_open( )** repeatedly. However, calling **sem_close( )** once for that semaphore will deallocate system resources associated with it. Any subsequent attempted use of that semaphore will return an error.

RETURNS

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

ERRNO

EINVAL
The semaphore descriptor is invalid or 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 causes an **EINVAL** error. Subsequent use of *sem* after destruction also causes 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.

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

ERRNO

EINVAL
The semaphore descriptor is invalid or 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 * _Restrict 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 by the time it is returned to the calling task.

If *sem* is locked, the value returned by **sem_getvalue( )** is either zero or a negative number whose absolute value represents the number of tasks waiting for the semaphore at some unspecified time during the call.

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

ERRNO

EINVAL
The semaphore descriptor or the *sval* pointer is invalid.

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 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* is ignored. Unnamed semaphores cannot be accessed from other RTPs.

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.

The *value* argument can only take up to 32 bits. 64 bit values will be truncated.

RETURNS

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

ERRNO

EINVAL
value exceeds **SEM_VALUE_MAX** or *sem* points to an invalid buffer.

ENOSPC
The semaphore cannot be initialized 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( )** using *name*, the task may reference the semaphore associated with *name* using the address returned by this call. The semaphore address returned may be used in subsequent calls to **sem_wait( )**, **sem_trywait( )**, and **sem_post( )**. This semaphore remains usable until it is closed by a successful call to **sem_close( )**. Multiple sem_open calls on the same named semaphore return the address of the same semaphore instance provided there have been no calls to **sem_unlink( )** for this semaphore, and till it is closed by a call to **sem_close( )**.

The *oflag* argument controls whether a new semaphore is created or merely accessed by the call to **sem_open( )**. The following flag bits may be set in *oflag*:

O_CREAT
: This flag to creates 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( )** creates a new semaphore. The **O_CREAT** option requires a third and fourth argument as follows: *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( )** fails if the semaphore name already exists. If **O_EXCL** is set, **O_CREAT** is not set, and the named semaphore does not exist, it is not created.

If the semaphore *name* begins with the forward-slash character, it is treated as a public semaphore. All RTPs can open their own references to this public semaphore by using its name *name*. If *name* does not begin with the forward-slash character, it is treated as a private semaphore and other RTPs cannot access it.

To determine whether a named semaphore already exists in the system, call **sem_open( )** with the flags **O_CREAT | O_EXCL**. If this **sem_open( )** call fails, the semaphore exists.

References to copies of the semaphore 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 the structure **sem_t**, or -1 (**SEM_FAILED**) if unsuccessful.

ERRNO

EEXIST
: **O_CREAT** and **O_EXCL** are set and the semaphore already exists

EINVAL
: *value* exceeds **SEM_VALUE_MAX** or 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 returns 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.

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

ERRNO **EINVAL**
The semaphore descriptor is invalid.

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 * _Restrict          sem,
    const struct timespec * _Restrict 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 remains 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.

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_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 remains 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.

RETURNS

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

ERRNO

EAGAIN
The semaphore is already locked.

EINVAL
The semaphore descriptor is invalid.

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.

RETURNS

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

ERRNO

ENAMETOOLONG
The semaphore name is 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 does 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 remains 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.

RETURNS

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

ERRNO

EINVAL
The semaphore descriptor is invalid.

EINTR
Signal received while blocking on the semaphore

SEE ALSO

semPxLib, **sem_trywait( )**, **sem_post( )**

setenv( )

NAME

setenv( ) – add or change an environment variable (POSIX)

SYNOPSIS

```
int setenv
    (
    const char * envVarName,   /* environment variable name */
    const char * envVarValue,  /* environment variable value */
    int          overwrite     /* if non-zero, change value when var exists
*/
    )
```

DESCRIPTION

This routine adds a new environment variable *envVarName* to the global environment if the variable does not already exist, or updates an existing variable if the value of the *overwrite* parameter is non-zero. If the *overwrite* parameter is left null, then existing environment variables are not modified.

An environment variable is a string with the following format: <variable name>=<variable value>. A variable name may not be **NULL**, the empty string or hold a "=" character.

RETURNS

0 for success, **ENOMEM** if the memory cannot be allocated for the string, **EINVAL** if the variable name is not valid.

ERRNO

N/A

SEE ALSO

setenv, **unsetenv( )**, **getenv( )**

setprlimit( )

NAME

setprlimit( ) – set process resource limits (syscall)

SYNOPSIS

```
int setprlimit
    (
    int              idtype,
    RTP_ID           id,
    int              resource,
    struct rlimit * rlp
    )
```

DESCRIPTION

none

RETURNS

0 on success, -1 on errors.

ERRNO

EFAULT
The address specified for rlp is invalid.

EINVAL
Invalid arguments, or operation failed.

SEE ALSO

ioLib

shm_open( )

NAME

shm_open( ) – open a shared memory object

SYNOPSIS

```
int shm_open
    (
    const char * name,   /* object name */
    int          oflag,  /* access control flag */
    mode_t       mode    /* permission mode */
    )
```

DESCRIPTION

The **shm_open( )** function establishes a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object.

The name argument points to a string naming a shared memory object. This argument must conform to one of the following two formats:

/*obj_name*
The name start with the slash (/) character, and must not contain any other slash characters. This is the more portable format. Note that multiple consecutive slash characters are treated as one.

/shm/*obj_name*
The name starts with the shmFs device name (/shm by default), followed by the separator slash (/) character, then followed by the object name. *obj_name* must not contain any slash characters.

If *name* does not conform to these rules, **shm_open( )** returns -1 and errno is set to **EINVAL**. The maximum length for *obj_name* (excluding the leading slash and terminating '\0' character) is 255. This value is returned by **pathconf( )** invoked with with _PC_NAME_MAX.

If successful, **shm_open( )** returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes.

The file status flags and file access modes of the open file description are according to the value of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the *fcntl.h* header. Applications specify exactly one of the first two values (access modes) below in the value of *oflag*:

O_RDONLY
Open for read access only.

O_RDWR
Open for read or write access.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT
If the shared memory object exists, this flag has no effect, except as noted under **O_EXCL** below. Otherwise, the shared memory object is created; the user and group ID of the shared memory object is set to a system default. The permission bits of the shared memory object is set to the value of the *mode* argument except those set in the file mode creation mask of the process. The *mode* argument is a bitwise inclusive OR of the read and write permissions defined in *sys/stat.h*. When bits in *mode* other than the file permission bits are set, they are masked out and ignored. The *mode* argument does not affect whether the shared memory object is opened for reading, for writing, or for both. A newly created shared memory object has an initial size of zero.

O_EXCL
If **O_EXCL** and **O_CREAT** are set, **shm_open( )** fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing **shm_open( )** naming the same shared memory object with **O_EXCL** and **O_CREAT** set. If **O_EXCL** is set and **O_CREAT** is not set, **O_EXCL** is ignored.

O_TRUNC
If the shared memory object exists, and it is successfully opened **O_RDWR**, the object is truncated to zero length and the mode and owner is unchanged by this function call. Using **O_TRUNC** with **O_RDONLY** returns -1 and errno is set to **EACCES**.

RETURNS

the lowest numbered unused file descriptor for the process, or -1 in case of error.

ERRNO

EACCES
The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or **O_TRUNC** is specified and write permission is denied.

EEXIST
O_CREAT and **O_EXCL** are set and the named shared memory object already exists.

EINTR
The **shm_open( )** operation was interrupted by a signal.

EINVAL
The **shm_open( )** operation is not supported for the given *name*.

EMFILE
Too many file descriptors are currently in use by this process.

ENAMETOOLONG
The length of the *name* argument exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.

ENOENT
O_CREAT is not set and the named shared memory object does not exist.

ENOSPC
There is insufficient space for the creation of the new shared memory object.

ENOSYS
The shared memory component is not supported.

SEE ALSO **shmLib**

---

shm_unlink( )

NAME **shm_unlink( )** – remove a shared memory object

SYNOPSIS

```
int shm_unlink
    (
    const char * name
    )
```

DESCRIPTION The **shm_unlink( )** function removes the name of the shared memory object named by the string pointed to by *name*. For the rules of constructing names of shared memory objects see the **shm_open( )** reference.

If one or more references to the shared memory object exist when the object is unlinked, the name is removed before **shm_unlink( )** returns, but the removal of the memory object contents are postponed until all open and map references to the shared memory object have been removed.

When an object remains alive due to existing references (open file descriptor, or mapped in memory) after the **shm_unlink( )** is called, reuse of the name subsequently causes **shm_open( )** to behave as if no shared memory object of this name exists. That means **shm_open( )** will fail if **O_CREAT** is not set, and will create a new shared memory object if **O_CREAT** is set.

RETURNS 0 in case of success, -1 otherwise

ERRNO

EACCES
Permission is denied to unlink the named shared memory object.

EINVAL
The **shm_open( )** operation is not supported for the given *name*.

ENAMETOOLONG
The length of the *name* argument exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.

ENOENT
The named shared memory object does not exist.

ENOSYS
The shared memory component is not supported.

SEE ALSO

shmLib

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. Parameter *signo* specifies the signal to operate on. Arguments *pAct* and *pOact* are pointers to sigaction structures. *pAct* describes the action to be taken when signal *signo* is received. If the *pAct* argument is not **NULL**, the response of the calling process to signal *signo* is altered as specified by the members of *pAct*. If argument *pOact* is not **NULL**, the action previously associated with signal *signo* is stored in the location pointed to by the *pOact*. If *pAct* is **NULL** but *pOact* is not, the current action associated with *signo* is returned. On the other hand, if *pOact* is **NULL** and *pAct* is not, the action associated with *signo* is changed as specified by *pAct* but the previously associated action is lost.

The sigaction structure has the following members -

Member	Meaning
sa_handler	Address of handler function having prototype void (*)(int)
sa_sigaction	Address of handler function having prototype void (*)(int, siginfo_t *, void *)

Member	Meaning
sa_mask	Additional set of signals to be blocked during execution of the signal handler.
sa_flags	Special flags to affect behavior of signal.

sa_handler and *sa_sigaction* are two different prototypes that the handler function can follow. Either can be used at any given time, but not both. *sa_handler* and *sa_sigaction* are members of a C union. In other words, they both occupy overlapped storage.

The *sa_flags* member consists of a set of flags defined as follows -

SA_NOCLDSTOP
Do not generate SIGCHLD when a child process stops or a stopped child continues.

SA_ONSTACK
If set and an alternate signal stack has been declared with **sigaltstack( )**, the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

SA_RESETHAND
If set, the action associated with *signo* is reset to **SIG_DFL** and the **SA_SIGINFO** flag shall be cleared on entry to the signal handler. However the SIGKILL and SIGTRAP signals cannot be automatically reset when delivered. In addition, if this flag is set, **sigaction( )** behaves as if the **SA_NODEFER** flag were also set.

SA_RESTART
This flag affects the behavior of interruptible functions (i.e. those specified to fail with errno set to **EINTR**). If this bit is set, and a function specified as interruptible is interrupted by this signal, the interrupted function shall restart and shall not fail with **EINTR** unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal shall fail with errno set to **EINTR**.

SA_SIGINFO
If this bit is clear and the signal *signo* is caught, the signal handler shall be entered using the *sa_handler* prototype (i.e. signo is the only argument to the handler). On the other hand if **SA_SIGINFO** is set and the *signo* is caught, the signal handler shall be entered using the *sa_sigaction* prototype.

SA_NOCLDWAIT
If set, and *signo* is SIGCHLD, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and **wait( )**, **waitid( )**, and **waitpid( )** shall fail and set errno to **ECHILD**. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to **SIG_IGN**.

SA_NODEFER
If set and *signo* is caught, *signo* shall not be added to the task's signal mask on entry to the signal handler unless it is included in sa_mask. Otherwise, *signo* shall always be added to the task's signal mask on entry to the signal handler.

The SIGKILL and SIGSTOP signals cannot be masked using this function.

The following is an example usage of the sigaction function to install a handler for signal SIGUSR1. In this example the *pOact* argument is **NULL** which means the original action associated with SIGUSR1 is lost.

```
#include <signal.h>

struct sigaction myAction;

void myHandler (int signo)
    {
    printf ("myHandler: receibed signal %d\n", signo);
    }

myAction.sa_handler = myHandler;
myAction.sa_flags = 0;
sigaction (SIGUSR1, &myAction, NULL);
```

RETURNS
0, or -1 if the signal number is invalid.

ERRNO
EINVAL
The *signo* is not a valid signal number.

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
0, or -1 if the signal number is invalid.

ERRNO **EINVAL**
The *signo* is not a valid signal number.

SEE ALSO **sigLib**

sigaltstack( )

NAME **sigaltstack( )** – set or get signal alternate stack context (syscall)

SYNOPSIS

```
int sigaltstack
  (
  const stack_t *ss,
  stack_t       *oss
  )
```

DESCRIPTION This routine allows an RTP task to define and examine the state of an alternate stack area on which signals are processed. If *ss* is non-zero, it specifies a pointer to and the size of a stack area on which to deliver signals, and informs the system whether the task is currently executing on that stack. When a signal's action indicates its handler should execute on the alternate signal stack (specified with a sigaction call), the system checks whether the task chosen to execute the signal handler is currently executing on that stack. If the task is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

The stack_t structure includes the following members:

int *ss_sp
long ss_size
int ss_flags

If ss is not **NULL**, it points to a structure specifying the alternate signal stack that will take effect upon successful return from **sigaltstack( )**. The ss_sp and ss_size members specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The ss_flags member specifies the new stack state and may be set to the following:

SS_DISABLE
The stack is to be disabled and ss_sp and ss_size are ignored. If **SS_DISABLE** is not set, the stack will be enabled.

If oss is not **NULL**, it points to a structure specifying the alternate signal stack that was in effect prior to the call to **sigaltstack( )**. The ss_sp and ss_size members specify the base and size of that stack. The ss_flags member specifies the stack's state, and may contain the following values:

SS_ONSTACK
The task is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the task is executing on it will fail.

SS_DISABLE
The alternate signal stack is currently disabled.

This is a POSIX specified routine.

RETURNS
OK (0), or **ERROR** (-1) on errors (see below).

ERRNO
EINVAL
The *ss* argument is not **NULL**, and the ss_flags member pointed to by *ss* contains flags other than **SS_DISABLE**.

ENOMEM
The size of the alternate stack area is less than MINSIGSTKSZ.

EPERM
An attempt was made to modify an active stack.

SEE ALSO
sigLib, **sigaction( )**

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 bit that is set in the 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

SYNOPSIS

```
int sigdelset
    (
    sigset_t       * pSet,  /* signal set to delete signal from */
    int              signo  /* signal to delete */
    )
```

DESCRIPTION This routine deletes the signal *signo* from the signal set *pSet*.

RETURNS 0, or -1 if the signal number is invalid.

ERRNO **EINVAL**
The *signo* is not a valid signal number.

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 0, or -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 0, or -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, 0 if it is not, or -1 if the test fails.

ERRNO **EINVAL**
The *signo* specified is not a valid signal number.

SEE ALSO **sigLib**

siglongjmp( )

NAME

siglongjmp( ) – perform non-local goto by restoring saved environment

SYNOPSIS

```
void siglongjmp
    (
    jmp_buf env,
    int     val
    )
```

DESCRIPTION

This routine restores the environment saved by the most recent invocation of the **sigsetjmp( )** routine that used the same **jmp_buf** specified in the argument *env*. The restored environment includes the program counter, thus transferring control to the **setjmp( )** caller.

The signal mask of the calling task will be restored if *env* was initialized by a call to **sigsetjmp( )** that specifed its *savemask* argument to be non-zero.

If there was no corresponding **sigsetjmp( )** call, or if the function containing the corresponding **sigsetjmp( )** routine call has already returned, the behavior of **siglongjmp( )** is unpredictable.

All accessible objects in memory retain their values as of the time **siglongjmp( )** was called, with one exception: local objects on the C stack that are not declared **volatile**, and have been changed between the **sigsetjmp( )** invocation and the **siglongjmp( )** call, have unpredictable values.

The **siglongjmp( )** function executes correctly in contexts of signal handlers and any of their associated functions (but not from interrupt handlers).

WARNING

Do not use **siglongjmp( )** or **sigsetjmp( )** from an ISR.

RETURNS

This routine does not return to its caller. Instead, it causes **sigsetjmp( )** to return *val*, unless *val* is 0; in that case **sigsetjmp( )** returns 1.

ERRNO

no errnos for this routine

SEE ALSO

longjmp, **sigsetjmp( )**

signal( )

NAME

signal( ) – specify the handler associated with a signal (POSIX)

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 by the calling process. If the value of *pHandler* is **SIG_DFL**, default action associated with that signal will be taken. If the value of *pHandler* is **SIG_IGN**, the signal will be ignored. Otherwise, *pHandler* must point to a function to be called when *signo* is received by the calling process.

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.

WARNING

This function is not reentrant. If more than one task in a given RTP can call this function, the calls should be made in a mutually exclusive manner, such as by calling **taskRtpLock( )** first.

RETURNS

The value of the previous signal handler, or **SIG_ERR**.

ERRNO

EINVAL
The specified *signo* is invalid.

SEE ALSO

sigLib, **sigaction( )**

sigpending( )

NAME

sigpending( ) – retrieve the set of pending signals (syscall)

SYNOPSIS

```
int sigpending
  (
  sigset_t * pSet
  )
```

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*.

This is a POSIX specified routine.

RETURNS

OK (0), or **ERROR** (-1) if the signal TCB cannot be allocated.

ERRNO

ENOMEM
Not enough memory to perform the operation.

SEE ALSO

sigLib

sigprocmask( )

NAME

sigprocmask( ) – examine and/or change the signal mask for an RTP (syscall)

SYNOPSIS

```
int sigprocmask
  (
  int how,
  const sigset_t * pSet,
  sigset_t * pOset
  )
```

DESCRIPTION

This routine allows the calling process to examine and/or change its signal mask. If *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 *pSset*.

This is a POSIX specified routine.

RETURNS

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

ERRNO

EINVAL
The *how* argument is invalid.

EFAULT
Invalid addresses for *pSet* or *pOset*.

SEE ALSO

sigLib, **sigsetmask( )**, **sigblock( )**

sigqueue( )

NAME

sigqueue( ) – send a queued signal to a RTP (POSIX)

SYNOPSIS

```
int sigqueue
    (
    pid_t              rtpId,
    int                signo,
    const union sigval value
    )
```

DESCRIPTION

The routine **sigqueue( )** sends the signal *signo* with the signal-parameter value *value* to the process *rtpId*. Any task in the target RTP that has unblocked *signo* can receive the signal.

RETURNS

OK (0), or **ERROR** (-1) if the RTP ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO

ESRCH
The RTP does not exist.

EINVAL
The *signo* specified is not a valid signal number.

EAGAIN
There is no resources to queue the signal.

SEE ALSO

sigLib, **rtpSigqueue( )**

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 bit that is set in the 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

SYNOPSIS

```
int sigsuspend
  (
  const sigset_t * pSet
  )
```

DESCRIPTION This routine suspends the calling task until delivery of a signal. While suspended, *pSet* is used as the set of masked signals. It is not possible to block signals that cannot be ignored. This is enforced by the system without causing an error to be indicated.

If the action is to terminate the process then **sigsuspend( )** shall never return. If the action is to execute a signal handler, then **sigsuspend( )** shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the **sigsuspend( )** call.

This is a POSIX specified routine.

NOTE Since the **sigsuspend( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**
A signal has interrupted the calling thread.

SEE ALSO **sigLib**

sigtimedwait( )

NAME **sigtimedwait( )** – wait for a signal

SYNOPSIS

```
int sigtimedwait
  (
  const sigset_t              *pSet,
```

```
siginfo_t                     *pInfo,
const struct timespec       *pTimeout
)
```

DESCRIPTION

The function **sigtimedwait( )** selects the pending signal from the set *pSet*. If multiple signals in *pSet* are pending, it awill 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 either one of the signals in *pSet* become pending, or 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 asynchronously 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.

This is a POSIX specified routine.

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. Or signal set has unsupported signals.

EFAULT
The pointers *pInfo* or *pTime* might be pointing to illegal address.

SEE ALSO **sigLib**

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 0, or -1 if the signal number is invalid or the signal TCB cannot be allocated.

ERRNO
EINVAL
EFAULT

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 **EINVAL** on failure.

ERRNO

EINVAL
Signal set has unsupported signals.

EINTR
The wait was interrupted by an unblocked, caught signal.

SEE ALSO

sigLib, **sigtimedwait( )**

sigwaitinfo( )

NAME

sigwaitinfo( ) – wait for signals (POSIX)

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 (i.e. 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 or caught signal.

EINVAL
Signal in set are unsupported signals.

EFAULT
The pointers *pInfo* or *pSet* might be pointing to illegal address.

SEE ALSO **sigLib**, **sigtimedwait( )**

sleep( )

NAME **sleep( )** – delay for a specified amount of time

SYNOPSIS

```
unsigned sleep
    (
    unsigned 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).

NOTE 64 bit value for the *secs* argument is not supported.

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( )**

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
    (
    const 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( )**

strtok_r( )

NAME **strtok_r( )** – break down a string into tokens (reentrant) (POSIX)

SYNOPSIS

```
char * strtok_r
    (
    char *       string,      /* string to break into tokens */
    const char * separators,  /* the separators */
    char **      ppLast       /* pointer to serve as string index */
    )
```

DESCRIPTION This routine considers the null-terminated string *string* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *separators*. The argument *ppLast* points to a user-provided pointer which in turn points to the position within *string* at which scanning should begin.

In the first call to this routine, *string* points to a null-terminated string; *separators* points to a null-terminated string of separator characters; and *ppLast* points to a **NULL** pointer. The function returns a pointer to the first character of the first token, writes a null character into *string* immediately following the returned token, and updates the pointer to which *ppLast* points so that it points to the first character following the null written into *string*. (Note that because the separator character is overwritten by a null character, the input string is modified as a result of this call.)

In subsequent calls *string* must be a **NULL** pointer and *ppLast* must be unchanged so that subsequent calls will move through the string *string*, returning successive tokens until no tokens remain. The separator string *separators* may be different from call to call. When no token remains in *string*, a **NULL** pointer is returned.

RETURNS A pointer to the first character of a token, or a **NULL** pointer if there is no token.

ERRNO Not Available

SEE ALSO **strtok_r**, **strtok( )**

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*.

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*.

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.

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.

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

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**

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 RTP's heap partition is stored in the RTP global variable **heapPartId**, 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**

sysAuxClkRateGet( )

NAME **sysAuxClkRateGet( )** – get the auxilary clock rate

SYNOPSIS
```
int sysAuxClkRateGet(void)
```

DESCRIPTION This routine returns the auxilary clock rate.

RETURNS The number of ticks per second of the auxilary clock.

ERRNO N/A

SEE ALSO **sysLib**, **tickGet( )**, the VxWorks programmer guides.

sysBspRev( )

NAME **sysBspRev( )** – get the BSP version and revision number

SYNOPSIS
```
char * sysBspRev(void)
```

DESCRIPTION This routine returns the BSP version and revision number

RETURNS A pointer to the BSP version/revision string.

ERRNO N/A

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

sysClkRateGet( )

NAME **sysClkRateGet( )** – get the system clock rate

SYNOPSIS
```
int sysClkRateGet(void)
```

DESCRIPTION This routine returns the system clock rate.

RETURNS The number of ticks per second of the system clock.

ERRNO N/A

SEE ALSO **sysLib**, **tickGet( )**, the VxWorks programmer guides.

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 logical memory

RETURNS The address of the top of logical memory.

ERRNO N/A

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

sysModel( )

NAME **sysModel( )** – get the model name of the CPU board

SYNOPSIS `char * sysModel(void)`

DESCRIPTION This routine returns the model name of the CPU board

RETURNS A pointer to the board name.

ERRNO N/A

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

sysPhysMemTop( )

NAME **sysPhysMemTop( )** – get the address of the top of physical memory

SYNOPSIS

```
char * sysPhysMemTop(void)
```

DESCRIPTION This routine returns the address of the top of physical memory

RETURNS The address of the top of physical memory.

ERRNO N/A

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

---

sysProcNumGet( )

NAME **sysProcNumGet( )** – get the processor number

SYNOPSIS

```
int sysProcNumGet(void)
```

DESCRIPTION This routine returns the processor number for the CPU board.

RETURNS The processor number for the CPU board.

ERRNO N/A

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

---

syscall( )

NAME **syscall( )** – invoke a system call using supplied arguments and system call number

SYNOPSIS

```
int syscall
    (
    int arg1,            /* first argument to system call */
    int arg2,            /* second argument to system call */
    int arg3,            /* etc. */
    int arg4,
    int arg5,
    int arg6,
    int arg7,
    int arg8,
    int scn              /* system call number */
    )
-----------------------------------------------------------------
```

```
|                |                        |                        |
|   reserved     |          Group#        |        Routine#        |
|                |                        |                        |
-----------------------------------------------------------------
31                                                              0
```

CAVEAT This routine has been provided for testing and diagnostic capabilities. Users are advised to be careful with the system call number argument in particular and other arguments in general, otherwise strange behaviours will result from calls being directed to an unexpected system call number.

RETURNS -1 (**ERROR**) when the system call handler reports an error condition, a value equal to or greater than zero otherwise.

ERRNOS As set by the system call handler.

SEE ALSO **rtpLib**, target/share/h/syscallUsrApi.def.template, target/share/h/syscallUsrApi.def.template, **target/h/syscall.h**

syscallGroupNumRtnGet( )

NAME **syscallGroupNumRtnGet( )** – return the number of routines in a system call group

SYNOPSIS

```
int syscallGroupNumRtnGet
    (
    int   syscallGroup,
    int * pNRtn
    )
```

DESCRIPTION This function takes a system call group number, and a pointer to an integer, and returns the number of routines present in that system call group *pNRtn*

If the system call group number is not present in the running system, it will return **ENOENT**.

RETURNS success or failure.

ERRNO **OK**

ENOENT
if system call not present

SEE ALSO **sysLib**

syscallGroupPresent( )

NAME

syscallGroupPresent( ) – check if a system call group is present from user mode

SYNOPSIS

```
int syscallGroupPresent
   (
   int    syscallGroup,
   char * buffer,
   int  * bufSize
   )
```

DESCRIPTION

This function takes a system call group number, a pointer to a buffer and a pointer to the length of the buffer. It will return the first **bufSize* bytes of the system call group name. If there isn't enough space for the system call group name, **ENOMEM** is returned. If the system call group number is not present in the running system, it will return **ENOENT**.

RETURNS

success or failure.

ERRNO

OK

ENOENT
if system call not present

ENOMEM
if there isn't enough buffer space for the full system call name.

SEE ALSO

sysLib

syscallInfo( )

NAME

syscallInfo( ) – get information on a system call from user mode

SYNOPSIS

```
int syscallInfo
    (
    int    syscallNum,
    char * buffer,
    int  * bufSize,
    int    type
    )
```

DESCRIPTION

This function takes a system call number, a pointer to a buffer and a pointer to the length of the buffer, and a specific sub-request. It will returns information about the system : the first **bufSize* bytes return the details.

If there isn't enough space for the information, **ENOMEM** is returned. If the system call number or group is not present in the running system, it will return **ENOENT**.

type can be:

KERN_SYSCALL_NAME
Returns the system call name of the requested system call num

KERN_SYSCALL_GROUP
Returns the system call group name of the requested group num

RETURNS
success or failure.

ERRNO
OK

ENOENT
if system call not present

ENOMEM
if there isn't enough buffer space for the full system call name.

SEE ALSO
sysLib

syscallNumArgsGet( )

NAME
syscallNumArgsGet( ) – return the number of arguments a system call takes

SYNOPSIS

```
int syscallNumArgsGet
   (
   int    syscallNum,
   int *  pNargs
   )
```

DESCRIPTION
This function takes a system call number, and a pointer to an integer, and returns the number of arguments that system call takens in *pNargs*

If the system call number is not present in the running system, it will return **ENOENT**.

RETURNS
success or failure.

ERRNO
OK

ENOENT
if system call not present

SEE ALSO
sysLib

syscallPresent( )

NAME **syscallPresent( )** – check if a system call is present from user mode

SYNOPSIS

```
int syscallPresent
   (
   int    syscallNum,
   char * buffer,
   int  * bufSize
   )
```

DESCRIPTION This function takes a system call number, a pointer to a buffer and a pointer to the length of the buffer. It will return the first **bufSize* bytes of the system call name. If there isn't enough space for the system call name, **ENOMEM** is returned. If the system call number is not present in the running system, it will return **ENOENT**.

RETURNS success or failure.

ERRNO **OK**

ENOENT
if system call not present

ENOMEM
if there isn't enough buffer space for the full system call name.

SEE ALSO **sysLib**

sysconf( )

NAME **sysconf( )** – get configurable system variables

SYNOPSIS

```
long sysconf
    (
    int name
    )
```

DESCRIPTION This routine allows an application to determine the current value of a system limit or know whether a feature is supported or not.

The *name* argument represents a system limit which value is to be returned, or a feature which support is to be queried. Note that, as -1 may be returned even for a successful case, the caller must first set the variable *errno* to 0, call **sysconf( )**, then check the content of *errno* if the function returned -1.

The supported system variables and corresponding names are listed in the table below (limits are between curly braces):

System Limit or Feature	Name Argument	Comments
{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX	
{AIO_MAX}	_SC_AIO_MAX	
{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX	
{ARG_MAX}	_SC_ARG_MAX	
_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO	Unsupported feature
_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO	
{ATEXIT_MAX}	_SC_ATEXIT_MAX	
_POSIX_BARRIERS	_SC_BARRIERS	Unsupported feature
{BC_BASE_MAX}	_SC_BC_BASE_MAX	Unsupported feature
{BC_DIM_MAX}	_SC_BC_DIM_MAX	Unsupported feature
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX	Unsupported feature
{BC_STRING_MAX}	_SC_BC_STRING_MAX	Unsupported feature
{CHILD_MAX}	_SC_CHILD_MAX	See note below
N/A	_SC_CLK_TCK	equivalent to **sysClkRateGet( )**
_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION	
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX	
_POSIX_CPUTIME	_SC_CPUTIME	Unsupported feature
{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX	
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX	Unsupported feature
_POSIX_FSYNC	_SC_FSYNC	
N/A	_SC_GETGR_R_SIZE_MAX	Unsupported feature
N/A	_SC_GETPW_R_SIZE_MAX	Unsupported feature
{HOST_NAME_MAX}	_SC_HOST_NAME_MAX	
{IOV_MAX}	_SC_IOV_MAX	
_POSIX_IPV6	_SC_IPV6	Unsupported feature
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL	Unsupported feature
{LINE_MAX}	_SC_LINE_MAX	Unsupported feature
{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX	
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES	
_POSIX_MEMLOCK	_SC_MEMLOCK	
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE	
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION	
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING	
_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK	
{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX	See note below
{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX	
{NGROUPS_MAX}	_SC_NGROUPS_MAX	See note below
{OPEN_MAX}	_SC_OPEN_MAX	See note below
{PAGE_SIZE}	_SC_PAGE_SIZE	See note below
{PAGESIZE}	_SC_PAGESIZE	See note below
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO	Unsupported feature

System Limit or Feature	Name Argument	Comments
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING	Unsupported feature
_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS	Unsupported feature
{RE_DUP_MAX}	_SC_RE_DUP_MAX	Unsupported feature
_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS	Unsupported feature
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS	
_POSIX_REGEXP	_SC_REGEXP	Unsupported feature
{RTSIG_MAX}	_SC_RTSIG_MAX	
_POSIX_SAVED_IDS	_SC_SAVED_IDS	Unsupported feature
{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX	See note below
{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX	
_POSIX_SEMAPHORES	_SC_SEMAPHORES	
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS	
_POSIX_SHELL	_SC_SHELL	Unsupported feature
{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX	
_POSIX_SPAWN	_SC_SPAWN	Unsupported feature
_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS	Unsupported feature
_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER	Unsupported feature
{SS_REPL_MAX}	_SC_SS_REPL_MAX	
{STREAM_MAX}	_SC_STREAM_MAX	
{SYMLOOP_MAX}	_SC_SYMLOOP_MAX	Unsupported feature. See note below.
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO	
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR	
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE	
_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME	Unsupported feature
{PTHREAD_DESTRUCTOR _ITERATIONS}	_SC_THREAD_DESTRUCTOR _ITERATIONS	
{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX	
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT	
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT	
_POSIX_THREAD_PRIORITY _SCHEDULING	_SC_THREAD_PRIORITY _SCHEDULING	
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED	Unsupported feature
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS	
_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER	
{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN	
{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX	See note below
_POSIX_THREADS	_SC_THREADS	
_POSIX_TIMEOUTS	_SC_TIMEOUTS	
{TIMER_MAX}	_SC_TIMER_MAX	See note below
_POSIX_TIMERS	_SC_TIMERS	
_POSIX_TRACE	_SC_TRACE	
_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER	
{TRACE_EVENT_NAME_MAX}	_SC_TRACE_EVENT_NAME_MAX	
_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT	Unsupported feature

System Limit or Feature	Name Argument	Comments
_POSIX_TRACE_LOG	_SC_TRACE_LOG	
{TRACE_NAME_MAX}	_SC_TRACE_NAME_MAX	
{TRACE_SYS_MAX}	_SC_TRACE_SYS_MAX	See note below
{TRACE_USER_EVENT_MAX}	_SC_TRACE_USER_EVENT_MAX	
{TTY_NAME_MAX}	_SC_TTY_NAME_MAX	
_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS	Unsupported feature
{TZNAME_MAX}	_SC_TZNAME_MAX	
_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32	Unsupported C99 programing environment
_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG	Only supported C99 programing environment
_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64	Unsupported C99 programing environment
_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG	Unsupported C99 programing environment
_POSIX_VERSION	_SC_VERSION	
_XBS5_ILP32_OFF32	_SC_XBS5_ILP32_OFF32	Unsupported feature
_XBS5_ILP32_OFFBIG	_SC_XBS5_ILP32_OFFBIG	Unsupported feature
_XBS5_LP64_OFF64	_SC_XBS5_LP64_OFF64	Unsupported feature
_XBS5_LPBIG_OFFBIG	_SC_XBS5_LPBIG_OFFBIG	Unsupported feature
_XOPEN_CRYPT	_SC_XOPEN_CRYPT	Unsupported feature
_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N	Unsupported feature
_XOPEN_LEGACY	_SC_XOPEN_LEGACY	Unsupported feature
_XOPEN_REALTIME	_SC_XOPEN_REALTIME	Support restricted to PSE52 profile
_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS	Support restricted to PSE52 profile
_XOPEN_SHM	_SC_XOPEN_SHM	Unsupported feature
_XOPEN_STREAMS	_SC_XOPEN_STREAMS	Unsupported feature
_XOPEN_UNIX	_SC_XOPEN_UNIX	Unsupported feature
_XOPEN_VERSION	_SC_XOPEN_VERSION	Support restricted to PSE52 profile
_POSIX2_C_BIND	_SC_2_C_BIND	
_POSIX2_C_DEV	_SC_2_C_DEV	
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM	Unsupported feature
_POSIX2_FORT_DEV	_SC_2_FORT_DEV	Unsupported feature
_POSIX2_FORT_RUN	_SC_2_FORT_RUN	Unsupported feature
_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF	Unsupported feature
_POSIX2_PBS	_SC_2_PBS	Unsupported feature
_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING	Unsupported feature
_POSIX2_PBS_CHECKPOINT	_SC_2_PBS_CHECKPOINT	Unsupported feature
_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE	Unsupported feature
_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE	Unsupported feature
_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK	Unsupported feature

System Limit or Feature	Name Argument	Comments
_POSIX2_SW_DEV	**_SC_2_SW_DEV**	
_POSIX2_UPE	**_SC_2_UPE**	Unsupported feature
_POSIX2_VERSION	**_SC_2_VERSION**	Unsupported feature
_POSIX_26_VERSION	**_SC_POSIX_VERSION**	Support POSIX .26

NOTE

A few values are handled in a specific way which might conflict with the POSIX standard but is necessary because of the constraints created by the PSE52 profile or the configurability of VxWorks:

_SC_CHILD_MAX
is not associated with the **CHILD_MAX** macro. The rational is that the maximum number of simultaneous RTP is not limited in VxWorks (i.e. it is limited only by the amount of memory), and also an application conforming to PSE52 must act as if **CHILD_MAX** = 0.

_SC_MQ_OPEN_MAX
is not associated with the **MQ_OPEN_MAX** macro. The rational
is that the maximum number of open message queue in a RTP is not limited in VxWorks (i.e. it is limited only by the amount of memory).

_SC_NGROUPS_MAX
is not associated with the **NGROUPS_MAX** macro. The rational is that the POSIX standard requires a minimum value for the **NGROUPS_MAX** macro which cannot be satisfied with the PSE52 conformance. VxWorks supports only one (1) group.

_SC_OPEN_MAX
is not associated with the **OPEN_MAX** macro because the maximum number of file descriptors for a RTP is a configurable value on VxWorks.

_SC_PAGE_SIZE and **_SC_PAGESIZE**
are not associated with, respectively, the **PAGE_SIZE** and PAGESIZE macros because the page size depends on the processor architecture and the minimum value required by the POSIX standard is meaningless in a VxWorks system.

_SC_THREAD_THREADS_MAX
is not associated with the **PTHREAD_THREADS_MAX** macro. The rational is that the maximum number of threads that can be created per RTP is not limited in VxWorks (i.e. it is limited only by the mount of memory).

_SC_SEM_NSEMS_MAX
is not associated with the **SEM_NSEMS_MAX** macro. The rational is that the maximum number of semaphore in a RTP is not limited in VxWorks (i.e. it is limited only by the amount of memory).

_SC_SS_REPL_MAX
is not associated with the **SS_REPL_MAX** macro. The rational is that the maximum number of replenishment events in a **SCHED_SPORADIC** thread is not limited in VxWorks (i.e. it is limited only by the amount of memory).

_SC_SYMLOOP_MAX
is not associated with the **SYMLOOP_MAX** macro. The rational is that VxWorks' file system framework does not support symbolic links, and also that the PSE52 conformance does not require support for symbolic links.

_SC_TIMER_MAX
is not associated with the **TIMER_MAX** macro. The rational is that the maximum number of timers in a RTP is not limited in VxWorks (i.e. it is limited only by the amount of memory).

_SC_TRACE_SYS_MAX
is not associated with a **TRACE_SYS_MAX** macro. The rational is that the maximum number of traces in a system is not limited in VxWorks (i.e. it is limited only by the amount of memory).

In all these situations it is deemed that calling **sysconf( )** is more appropriate than using a macro.

RETURNS

The current value for the variable or -1. Supported features are indicated by a returned value equal or greater than 0. If -1 is returned and *errno* has not been modified, the variable has an indefinite limit or corresponds to an unsupported feature. If -1 is returned and *errno* has been changed to **EINVAL** the value passed as *name* is invalid.

ERRNO

EINVAL
when the value of the *name* argument is not valid.

SEE ALSO

sysconf

sysctl( )

NAME

sysctl( ) – get or set the the values of objects in the sysctl tree (syscall)

SYNOPSIS

```
int sysctl
    (
    int *     pName,
    u_int     nameLen,
    void *    pOld,
    size_t *  pOldLen,
    void *    pNew,
    size_t    newLen
    )
```

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.

EXAMPLE

```
#include "vxWorks.h"
#include "sys/sysctl.h"

#define BSP_REV_SIZE 256
#define OID_LEN  3

char bspRev[BSP_REV_SIZE];
int  bufSize = BSP_REV_SIZE;
int  mib[OID_LEN];

/* Fill out the three components of the sysctl OID */

mib[0] = CTL_HW;
mib[1] = HW_BSP;
mib[2] = HW_BSP_REVISION;

/* Make a system call to read BSP revision info */

if (sysctl (mib, OID_LEN, (void *)&bspRev, (size_t *)&bufSize,
            NULL, 0) == 0)
    printf ("BSP revision = %s\n", bspRev);
else
    printf ("sysctl failed %d\n", errno);
```

RETURNS

0 upon success, -1 if an error occurred.

ERRNO

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.

ENOSYS
The component **INCLUDE_SC_SYSCTL** has not been included in the kernel.

SEE ALSO

sysctlLib

taskActivate( )

NAME

taskActivate( ) – activate a task that has been created without activation

SYNOPSIS

```
STATUS taskActivate
    (
    int tid  /* task ID of task to activate */
    )
```

DESCRIPTION

This routine activates tasks created by the library routines **taskCreate( )** or **taskOpen( )**, or by tasks created by the **_taskOpen( )** system call. A task created by **taskOpen( )** or **_taskOpen( )** with the **VX_TASK_NOACTIVATE** option bit specified is not activated when created.

A task that has not been activated is ineligible for CPU allocation by the scheduler.

The **taskSpawn( )** routine is built from **taskCreate( )** and **taskActivate( )**. Tasks created by **taskSpawn( )** do not require explicit task activation.

RETURNS

OK, or **ERROR** if the task cannot be activated.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to activate a task in another RTP.

SEE ALSO

taskLib, **taskCreate( )**, **taskOpen( )**

taskClose( )

NAME

taskClose( ) – close a task

SYNOPSIS

```
STATUS taskClose
    (
    int tid  /* task to close */
    )
```

DESCRIPTION

This routine closes a task. It invalidates *tid* and decrements the task reference counter.

This routine does not delete a task. **taskDelete( )** should be called to terminate and delete a task.

RETURNS

OK, or **ERROR** if *tid* is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR
The specified task ID is invalid.

SEE ALSO

taskLib, **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.

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 */
        }
    }
}
```

RETURNS

OK or **ERROR** if the task ID is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskUtilLib, **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.

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);
   }
```

RETURNS

OK, or **ERROR** if the task ID or affinity is invalid.

ERRNO

S_taskLib_ILLEGAL_OPERATION

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskUtilLib, **taskCpuAffinityGet( )**, **vxCpuConfiguredGet( )**, **cpuset**

taskCreate( )

NAME

taskCreate( ) – allocate and initialize a task without activation

SYNOPSIS

```
int taskCreate
    (
    char *  name,       /* name of new task */
    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 a new **private** task with a specified priority and options. The memory for the stacks and task control block is dynamically allocated. Activate the newly created task by invoking **taskActivate( )**.

To create *and* activate a task, use the **taskSpawn( )** routine instead of **taskCreate( )**. To create a **public** task, use the general purpose **taskOpen( )** routine.

See **taskSpawn( )** for an explanation of all the parameters.

RETURNS

Task ID, or **NULL** if there is not enough memory or if the task cannot be created.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP to spawn the task.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the task handle.

S_taskLib_ILLEGAL_PRIORITY
A priority outside the range 0 to 255 was specified.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to specify an illegal location for the user stack.

S_taskLib_ILLEGAL_OPTIONS
The operation attempted to specify an unsupported option.

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified

S_objLib_OBJ_INVALID_ARGUMENT
name buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

SEE ALSO **taskLib**, **taskSpawn( )**, **taskOpen( )**, **taskActivate( )**

---

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. The new routine should be declared as follows:

```
STATUS createHook
    (
    int tid     /* task ID of new task */
    )
```

The create hook routine is executed in the context of the task invoking the task creation primitive. If any create hook routine returns **ERROR**, the task creation sequence is aborted; **taskOpen( )** and **taskCreate( )** return **NULL**, while **taskSpawn( )** returns **ERROR**.

RETURNS **OK**, or **ERROR** if the table of task create routines is full.

ERRNO **S_taskLib_TASK_HOOK_TABLE_FULL**
The task create hook table is full.

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 **S_taskLib_TASK_HOOK_NOT_FOUND**
The specified create hook was not found in the table.

SEE ALSO **taskHookLib**, **taskCreateHookAdd( )**

taskCtl( )

NAME **taskCtl( )** – perform a control operation against a task (system call)

SYNOPSIS

```
STATUS taskCtl
    (
    int             tid,
    VX_TASK_CTL_CMD command,
    void *          pArg,
    UINT *          pArgSize
    )
```

DESCRIPTION The **taskCtl( )** system call performs the requested *command* against a task specified by *tid*. The following is a description of the supported commands:

VX_TASK_CTL_ACTIVATE
Activate the specified task. The arguments *pArg* and *pArgSize* are not used for this command. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_SUSPEND
Suspend the specified task. The arguments *pArg* and *pArgSize* are not used for this command. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_RESUME
Resume the specified task. The arguments *pArg* and *pArgSize* are not used for this command. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_RESTART
Restart the specified task. The arguments *pArg* and *pArgSize* are not used for this command. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_PRI_NORMAL_GET
Get the specified task's normal priority. the priNormal is the "normal" assigned priority of the task assigned either at task creation time or with a call to **taskPrioritySet( )**. A task executes at its normal assigned priority unless priority inheritance has occurred. *pArg* is an integer pointer (int *) to the memory location to receive the priority. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (int *)**. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_PRIORITY_GET
Get the specified task's priority. The argument *pArg* is an integer pointer (int *) to the memory location to receive the priority. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (int *)**. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_PRIORITY_SET
Set the specified task's priority. The argument *pArg* is an integer pointer (int *) to the task's new priority. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (int *)**. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_VERIFY
Verify that the specified task identifier is valid. The arguments *pArg* and *pArgSize* are not used for this command. A return value of **OK** indicates that the task identifier is valid.

VX_TASK_CTL_VAR_ADD
Add a task variable to the specified task. The argument *pArg* is a pointer to a **VX_TASK_CTL_VAR_CMD**:

```
typedef struct vx_task_ctl_var_cmd
    {
    int *pVariable;   /* pointer to variable to be switched for task */
    int  value;       /* new value of task variable */
    } VX_TASK_CTL_VAR_CMD;
```

The **pVariable** field is set to the address of the task variable to add. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (VX_TASK_CTL_VAR_CMD**). This command cannot be issued against a task residing outside the current RTP.

This command is not available on SMP systems.

VX_TASK_CTL_VAR_DELETE

Delete a task variable from the specified task. The argument *pArg* is a pointer to a **VX_TASK_CTL_VAR_CMD** struct (described above). The **pVariable** field is set to the address of the task variable to delete. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (VX_TASK_CTL_VAR_CMD**). This command cannot be issued against a task residing outside the current RTP.

This command is not available on SMP systems.

VX_TASK_CTL_VAR_GET

Get 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. The argument *pArg* is a pointer to a **VX_TASK_CTL_VAR_CMD** struct (described above). The **pVariable** field is set to the address of the task variable whose value is to be read. The system call writes the value of the task variable into the **value** field. *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (VX_TASK_CTL_VAR_CMD**). This command cannot be issued against a task residing outside the current RTP.

This command is not available on SMP systems.

VX_TASK_CTL_VAR_SET

Set the private value of a task variable for the specified task. The specified task is usually not the calling task, which can set its private value by directly modifying the variable. The argument *pArg* is a pointer to a **VX_TASK_CTL_VAR_CMD** struct (see above). The **pVariable** field is set to the address of the task variable whose value is to be set. The value of the task variable is specified in the **value** field. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (VX_TASK_CTL_VAR_CMD**). This command cannot be issued against a task residing outside the current RTP.

This command is not available on SMP systems.

VX_TASK_CTL_TASK_EXIT

Exit the calling task without terminating the RTP. The *tid* parameter is ignored. The argument *pArg* is an integer pointer to the task's exit code. The exit code is passed to the **exit( )** routine provided by the Dinkum standard C library. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (int *)**.

VX_TASK_CTL_UTCB_SET
Register the pointer to the user-level version of the task control block for the specified task. The argument *pArg* is a pointer to the user-level task control block that will be registered. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (void *)**. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_UTCB_GET
Retrieve the pointer to the user-level version of the task control block for the specified task. The argument *pArg* is a pointer to a pointer to the user-level task control block that will be retrieved. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (void *)**. This command cannot be issued against a task residing outside the current RTP.

VX_TASK_CTL_EXIT_REGISTER
Register the address of a function that will be executed when the **main** routine, that is, the *entryPt* parameter specified in the **_taskOpen( )** system call, returns. The registration affects all tasks created in the current RTP, as the *tid* parameter is ignored. The argument *pArg* is a pointer to the function to be registered. The argument *pArgSize* is an unsigned integer pointer (UINT *) to a memory location containing the value **sizeof (void *)**. This command cannot be issued against a task residing outside the current RTP.

RETURNS

OK if the requested operation completes successfully, otherwise **ERROR**.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to perform a control command against a task in another RTP.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to perform the requested control command.

S_taskLib_ILLEGAL_PRIORITY
An illegal task priority is specified for the **VX_TASK_CTL_PRIORITY_SET** command.

S_taskLib_TASK_VAR_NOT_FOUND
Can not find the specified task variable for the **VX_TASK_CTL_VAR_DELETE**, **VX_TASK_CTL_VAR_GET**, or **VX_TASK_CTL_VAR_SET** control commands.

S_objLib_OBJ_INVALID_ARGUMENT
In commands in which *pArg* is needed, like **VX_TASK_CTL_VAR_GET** and **VX_TASK_CTL_PRIORITY_SET**, the buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden; Or it does belong to this RTP task but can not be read or written due to access control. In commands in which *pArg* is a data structure which carries buffer pointers, like **VX_TASK_CTL_VAR_GET** and **VX_TASK_CTL_VAR_SET**, those buffers, are not valid in memory addresses; Or valid but do not belong to the calling RTP task, so access is forbidden. e.g., an RTP task's auto

variables do not belong to another task in the same RTP. Or they do belong to the calling RTP task but the needed accesses, read, write or both, are not allowed.

SEE ALSO
taskLib

taskDelay( )

NAME
taskDelay( ) – delay calling task from executing (system call)

SYNOPSIS
```
STATUS taskDelay
   (
   int ticks
   )
```

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.

RETURNS
OK, or **ERROR** if the calling task receives a signal that is not blocked or ignored.

ERRNO
EINTR
The calling task received a signal that was not blocked or ignored during the delay.

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 any other memory resources including the task control block. Upon deletion, all routines specified

by **taskDeleteHookAdd( )** are called in the context of the deleting task. This routine is the companion routine to **taskSpawn( )** and **taskCreate( )**.

Tasks that reside outside the current RTP cannot be deleted.

RETURNS **OK**, or **ERROR** if the task cannot be deleted.

ERRNO

S_objLib_OBJ_DELETED
The specified task was deleted by a higher priority task.

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to delete a task in another RTP, or the task was created using a direct call to the **_taskOpen( )** system call and cannot be deleted using **taskDelete( )**.

SEE ALSO **taskLib**, **taskDeleteHookAdd( )**, **taskSpawn( )**, **taskCreate( )**

---

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( )** are called in the context of the deleting task.

Tasks that reside outside the current RTP cannot be deleted.

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 RTP in an unstable state or lead to deadlock.

The system does not protect against simultaneous **taskDeleteForce( )** calls. Such a situation could leave the system in an unstable state.

RETURNS **OK**, or **ERROR** if the task cannot be deleted.

ERRNO

S_objLib_OBJ_DELETED
The specified task was deleted by a higher priority task.

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to delete a task in another RTP, or the task was created using a direct call to the **_taskOpen( )** system call and cannot be deleted using **taskDeleteForce( )**.

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. The new routine should be declared as follows:

```
STATUS deleteHook
    (
    int tid     /* task ID of deleted task */
    )
```

If a task delete hook returns **ERROR** during **taskDelete( )**, the deletion operation is not aborted, but **taskDelete( )** returns **ERROR**. However, if a task delete hook returns **ERROR** during **taskRestart( )**, the restart operation is aborted and **taskRestart( )** returns **ERROR**.

RETURNS **OK**, or **ERROR** if the table of task delete routines is full.

ERRNO **S_taskLib_TASK_HOOK_TABLE_FULL**
The task delete hook table is full.

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 **S_taskLib_TASK_HOOK_NOT_FOUND**
The specified delete hook was not found in the table.

SEE ALSO **taskHookLib**, **taskDeleteHookAdd( )**

---

taskExit( )

NAME **taskExit( )** – exit a task

SYNOPSIS

```
void taskExit
    (
    int code
    )
```

DESCRIPTION This routine is called by a task to terminate itself, to cease to exist as a task. It is called implicitly when the **main** routine of a spawned task exits. The *code* parameter is stored in the task control block for possible use by the delete hooks or for post-mortem debugging.

The **taskExit( )** function differs from the **exit( )** function in that invoking **exit( )** causes the entire RTP to terminate. The **taskExit( )** function differs from the **taskDelete( )** function in that **taskExit( )** allows you to set the code parameter to be examined after the task exits.

RETURNS N/A (since this function never returns)

ERRNO N/A (since this function never returns)

SEE ALSO **taskLib**, **taskDelete( )**, **exit( )** (Dinkum standard C library)

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 (that is, the user specified a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (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**

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.

RETURNS

The task ID of the calling task, or **ERROR** if the ID could not be determined.

ERRNO

S_taskLib_NO_TCB
The current task was created using a direct call to the **_taskOpen( )** system call and the ID cannot be determined by **taskIdSelf( )**.

SEE ALSO

taskLib, **taskSafe( )**

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.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**
The *tid* parameter is an invalid task ID.

SEE ALSO **taskLib**

taskInfoGet( )

NAME **taskInfoGet( )** – get information about a task

SYNOPSIS

```
STATUS taskInfoGet
    (
    int         tid,
    TASK_DESC * pTaskDesc
    )
```

DESCRIPTION This routine gets information about task *tid* and copies it to the **TASK_DESC** structure pointed to by the *pTaskDesc* argument.

TASK_DESC is defined in **taskLibCommon.h** as follows:

```
typedef struct                  /* TASK_DESC - information structure */
    {
    int          td_id;            /* task id                          */
    int          td_priority;      /* task priority                    */
    int          td_status;        /* task status                      */
    int          td_options;       /* task option bits (see below)     */
    FUNCPTR      td_entry;         /* original entry point of task     */
    char *       td_sp;            /* saved stack pointer              */
    char *       td_pStackBase;    /* the bottom of the stack          */
    char *       td_pStackEnd;     /* the actual end of the stack      */
    int          td_stackSize;     /* size of stack in bytes           */
```

```
int          td_stackCurrent;   /* current stack usage in bytes     */
int          td_stackHigh;      /* maximum stack usage in bytes     */
int          td_stackMargin;    /* current stack margin in bytes    */
int          td_errorStatus;    /* most recent task error status    */
int          td_delay;          /* delay/timeout ticks              */
EVENTS_DESC  td_events;         /* VxWorks events information       */
char         td_name [VX_TASK_NAME_LENGTH+1];  /* name of task      */
RTP_ID       td_rtpId;          /* RTP owning the task              */
int          td_excStackSize;   /* size of exception stack in bytes */
char *       td_pExcStackBase;  /* exception stack base             */
char *       td_pExcStackEnd;   /* exception stack end              */
char *       td_pExcStackStart; /* exception stack start            */
int          td_excStackHigh;   /* exception stack max usage        */
int          td_excStackMargin; /* exception stack margin           */
int           td_excStackCurrent;/* current exc stack usage in bytes */
} TASK_DESC;
```

Many of the members in the above structure are either not used when this routine is called from a user task or they have restrictions:

td_entry, *td_pStackBase*, *td_pStackEnd* are **NULL** if *tid* is a kernel task or a task residing in another RTP. *td_sp* is **NULL** if *tid* is a kernel task or a task residing in another RTP. Furthermore, if *tid* is a user task in a system call, *td_sp* is set to the user stack pointer at the time of the system call. If *tid* is in a system call, *td_stackCurrent* and *td_stackMargin* reflect the user stack usage at the time of the system call. Exception stack usage made during the system call is not included in these figures. *td_pExcStackBase*, *td_pExcStackEnd* and *td_pExcStackStart* are always **NULL**.

WARNING

The information provided by this routine is a snap shot of the *tid*'s state at the time of the call. By the time this routine returns, the *tid*'s state may have changed and therefore one must not make assumptions regarding the period of time for which the information provided continues to represent reality. Furthermore, in order to ensure coherency of the information, this routine needs to lock interrupts for periods of time. It therefore needs to be used judiciously so as to limit negative impact on system performance.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNOS

Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
ID is invalid.

SEE ALSO

taskInfo

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 Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
ID is invalid.

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 Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
ID is invalid.

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 Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
ID is invalid.

SEE ALSO **taskInfo**

taskKill( )

NAME **taskKill( )** – send a signal to a RTP task (syscall)

SYNOPSIS

```
int taskKill
  (
  int taskId;
  int signo;
  )
```

DESCRIPTION This routine sends a signal *signo* to a RTP task specified by *taskId*. This API can also be used to send signals to public tasks in other RTP's.

RETURNS **OK** (0), or **ERROR** (-1) if the task Id or signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **kill( )**, **rtpKill( )**

taskName( )

NAME **taskName( )** – get the name of a task residing in the current RTP

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. The specified task ID must reside in the same RTP as the calling task, otherwise **NULL** is returned. To obtain the name of task in another RTP, use the **taskNameGet( )** function.

RETURNS A pointer to the task name, or **NULL** if the task ID is invalid or the task resides in another RTP.

ERRNOS N/A

SEE ALSO **taskInfo**, **taskNameGet( )**

taskNameGet( )

NAME **taskNameGet( )** – get the name of any task

SYNOPSIS

```
STATUS taskNameGet
    (
    int    tid,
    char * pBuf,
    int    bufSize
    )
```

DESCRIPTION This routine copies the name string of the specified task *tid* into the caller-provided buffer *pBuf*. The size of the buffer is specified by the *bufSize* argument. If the buffer size is less than or equal to the name length, the buffer will not be null-terminated.

Unlike **taskName( )**, this function can be used to obtain the name of any task in the system. However, the **taskName( )** routine provides better performance to obtain the name of a task in the same RTP as the caller.

RETURNS **OK** if task name is copied successfully, or **ERROR** if the task ID is invalid.

ERRNO **S_objLib_OBJ_NAME_TRUNCATED**
The supplied name buffer is too small. A truncated name has been returned.

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

SEE ALSO **taskInfo**, **taskName( )**

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.

RETURNS The task ID, or **ERROR** if the task is not found.

ERRNO **S_taskLib_NAME_NOT_FOUND**
no task is found for the specified task name.

SEE ALSO **taskInfo**

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,  /* location of execution stack */
    int          stackSize,   /* execution stack size */
    void *       context,     /* context value */
    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

The **taskOpen( )** API is the most general purpose task creation routine. It can also be used to obtain a handle to an already existing task, typically a public task in another RTP. It searches the task name space for a matching task. If a matching task is found, it returns an object handle to 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.

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_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 user stack area. 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
The context value assigned to the created task. This value is not actually used by VxWorks. Instead, the context value is available for OS extensions to implement such facilities as object permissions.

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 **taskExit( )** 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.

RETURNS
The task ID, or **NULL** if unsuccessful.

ERRNO
S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP 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 an illegal location for the user stack.

S_taskLib_ILLEGAL_OPTIONS
The operation attempted to specify an unsupported option.

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified.

S_objLib_OBJ_HANDLE_TBL_FULL
There is no space in the RTP object handle table for the task handle.

S_objLib_OBJ_INVALID_ARGUMENT
An invalid option was specified in the *mode* argument or *name* is invalid. *name* buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control. *pStackBase* is provided, not **NULL**, it has the same problem as *name* buffer above; Or it can not be neither written nor read due to access control.

S_objLib_OBJ_OPERATION_UNSUPPORTED
The operation attempted to create an unamed public task.

S_objLib_OBJ_NOT_FOUND
The **OM_CREATE** flag was not set in the *mode* argument and a task matching *name* was not found.

SEE ALSO **taskLib**, **taskSpawn( )**, **taskCreate( )**, **taskActivate( )**, **taskClose( )**

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_PRIVATE_ENV
This task includes private environment support (see **envLib**).

VX_NO_STACK_FILL
This task does not fill the stack with 0xee for use by **checkstack( )**.

VX_TASK_NOACTIVATE
This task is not activated during **taskOpen( )**.

For definitions, see **taskLib.h**.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

ERRNOS Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
ID is invalid.

SEE ALSO **taskInfo**

taskPriNormalGet( )

NAME **taskPriNormalGet( )** – examine the normal priority of a task

SYNOPSIS

```
STATUS taskPriNormalGet
    (
    int   tid,         /* task ID */
    int * pPriNormal   /* return normal priority here */
    )
```

DESCRIPTION This routine determines the normal priority of a specified task. The normal priority is copied to the integer pointed to by *pPriNormal*. The priNormal is the "normal" assigned priority of the task assigned either at task creation time or with a call to **taskPrioritySet( )**. A task executes at its normal assigned priority unless priority inheritance has occurred.

RETURNS **OK**, or **ERROR** if the normal priority could not be obtained.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to obtain the priority of a task in another RTP.

S_objLib_OBJ_INVALID_ARGUMENT
pPriNormal buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., *pPriNormal* is an auto variable in a RTP task other than the task which calls taskPriorityGet. Or it does belong to this RTP task but can not be written due to access control.

SEE ALSO **taskLib**, **taskPrioritySet( )**, **taskPriorityGet( )**

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 priority could not be obtained.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to obtain the priority of a task in another RTP.

S_objLib_OBJ_INVALID_ARGUMENT
pPriority buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., *pPriority* is an auto variable in a RTP task other than the task which calls taskPriorityGet. Or it does belong to this RTP task but can not be written due to access control.

SEE ALSO

taskLib, **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 semphore 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.

RETURNS

OK, or **ERROR** if the priority cannot be changed.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to change the priority of a task in another RTP.

S_taskLib_ILLEGAL_PRIORITY
An illegal task priority was specified.

SEE ALSO **taskLib**, **taskPriorityGet( )**

taskRaise( )

NAME **taskRaise( )** – send a signal to the calling task

SYNOPSIS

```
int taskRaise
    (
    int signo    /* signal to send to caller's task */
    )
```

DESCRIPTION This routine sends the signal *signo* to the calling task.

RETURNS 0, or -1 if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **raise( )**, **rtpRaise( )**

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 "RestartTask" task.

Tasks that reside outside the current RTP cannot be restarted.

WARNING The initial task of an RTP cannot be restarted. This is because the initial task is involved with the instantiation of the RTP operating environment which has not be designed to be restartable.

NOTE If the task has modified any of its start-up parameters, the restarted task starts with the changed values.

RETURNS **OK**, or **ERROR** if the task ID is invalid or the task could not be restarted.

ERRNO

S_objLib_OBJ_DELETED
The specified task was destroyed by a higher priority task.

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to restart a task in another RTP, or a task that was created using a direct call to the **_taskOpen( )** system call and cannot be restarted using **taskRestart( )**.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to restart the task.

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. Thus suspended, delayed tasks remain suspended until their delays expire, and suspended, pended tasks remain pended until they unblock.

Tasks that reside outside the current RTP cannot be resumed.

RETURNS **OK**, or **ERROR** if the task cannot be resumed.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to resume a task in another RTP.

SEE ALSO **taskLib**

taskRtpLock( )

NAME

taskRtpLock( ) – disable task rescheduling

SYNOPSIS

```
STATUS taskRtpLock (void)
```

DESCRIPTION

This routine disables task context switching within an RTP. Invoking this function prevents other tasks in the same RTP from preempting the calling task. The calling task becomes the only task in the RTP 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 **taskRtpUnlock( )**; 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 **taskRtpUnlock( )** has been called as many times as **taskRtpLock( )**.

A **semTake( )** is preferable to **taskRtpLock( )** as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the RTP.

No primitives are provided in the RTP space for globally locking the scheduler as is done in the kernel by **taskLock( )**. As a result tasks in other RTPs may preempt a task locked with **taskRtpLock( )**. If exclusion between tasks in different RTPs is required, use a public semaphore in place of **taskRtpLock( )**.

SMP CONSIDERATIONS

This API is not supported for VxWorks SMP. Any usages of this API in an application for VxWorks SMP will error with a message (i.e. by default, terminated the RTP application) and an ED&R log will be generated.

Users are encouraged to utilized other synchronization mechanisms, such as semaphores or atomic operators, for their SMP application.

SMP CONSIDERATIONS

This routine is not suported in SMP. If it is called in SMP then **ERROR** will be returned.

RETURNS

OK or **ERROR**.

ERRNO

S_taskLib_NO_TCB
The current task was created using a direct call to the **_taskOpen( )** system call and cannot be locked using **taskRtpLock( )**.

SEE ALSO

taskLib, **taskRtpUnlock( )**, **taskSafe( )**, **semTake( )**

taskRtpUnlock( )

NAME

taskRtpUnlock( ) – enable task rescheduling

SYNOPSIS

```
STATUS taskRtpUnlock (void)
```

DESCRIPTION

This routine decrements the preemption lock count. Typically this call is paired with **taskRtpLock( )** and concludes a critical section of code. Preemption will not be unlocked until **taskRtpUnlock( )** has been called as many times as **taskRtpLock( )**. When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

SMP CONSIDERATIONS

This API is not supported for VxWorks SMP. Any usages of this API in an application for VxWorks SMP will error with a message (i.e. by default, terminated the RTP application) and an ED&R log will be generated.

Users are encouraged to utilized other synchronization mechanisms, such as semaphores or atomic operators, for their SMP application.

SMP CONSIDERATIONS

This routine is not suported in SMP. If it is called in SMP then **ERROR** will be returned.

RETURNS

OK or **ERROR**.

ERRNO

S_taskLib_NO_TCB
The current task was created using a direct call to the **_taskOpen( )** system call and cannot be unlocked using **taskRtpUnlock( )**.

SEE ALSO

taskLib, **taskRtpLock( )**

taskSafe( )

NAME

taskSafe( ) – make the calling task safe from deletion

SYNOPSIS

```
STATUS taskSafe (void)
```

DESCRIPTION

This routine protects the calling task from deletion by other tasks in the same RTP. A task residing in another RTP can still delete the current RTP (and thus delete the calling task).

Tasks that attempt to delete a protected task block until the task is made unsafe using **taskUnsafe( )**. When a task becomes unsafe, the deleter is 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, or **ERROR** if unable to make the task safe from deletion.

ERRNO
S_taskLib_NO_TCB
The current task was created using a direct call to the **_taskOpen( )** system call and cannot be made safe using **taskSafe( )**.

SEE ALSO
taskLib, **taskUnsafe( )**

---

taskSigqueue( )

NAME
taskSigqueue( ) – send a queued signal to a RTP task

SYNOPSIS
```
int taskSigqueue
    (
    int                taskId,
    int                signo,
    const union sigval value
    )
```

DESCRIPTION
This routine sends the signal *signo* with the signal-parameter value *value* to the RTP task *taskId*. This API can also be used to send signals to public tasks in other RTP's.

RETURNS
OK (0), or **ERROR** (-1) if the task handle or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO
EINVAL
EAGAIN

SEE ALSO
sigLib

taskSpawn( )

NAME

taskSpawn( ) – spawn a task

SYNOPSIS

```
int taskSpawn
    (
    char *   name,       /* name of new task */
    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 **private** task with a specified priority and options. The memory for the stacks and task control block is dynamically allocated.

To create but not activate a task, use the **taskCreate( )** routine instead. To create a **public** task, use the general purpose **taskOpen( )** routine.

A description of the **taskSpawn( )** arguments follows:

name

A task may be given a name as a debugging aid. This name appears in various kernel shell facilities such as **i( )**. The name may be of arbitrary length and content. If the task name is specified as **NULL**, an ASCII name is given to the task of the form **t***n* where *n* is a number which increments as tasks are spawned. Task names are not unique.

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_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_NO_STACK_PROTECT	do not provide overflow/underflow stack protection, stack remains executable

stackSize
The size in bytes of the execution stack area. The API returns **ERROR** if a negative stack size is passed as stackSize argument.

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.

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 **taskExit( )** 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.

RETURNS
The task ID, or **ERROR** if memory is insufficient or the task cannot be created.

ERRNO
S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory in the kernel or RTP 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 an illegal location for the user stack.

S_taskLib_ILLEGAL_OPTIONS
The operation attempted to specify an unsupported option.

S_taskLib_ILLEGAL_STACK_INFO
An invalid stack size has been specified.

S_objLib_OBJ_INVALID_ARGUMENT
name buffer, other than **NULL**, is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

SEE ALSO
taskLib, **taskCreate( )**, **taskOpen( )**, **taskActivate( )**

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 until resumed.

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.

As a synchronization mechanism, this facility should be rejected in favor of the more general semaphore facility.

Tasks that reside outside the current RTP cannot be suspended.

RETURNS

OK, or **ERROR** if the task cannot be suspended.

ERRNO

S_objLib_OBJ_ID_ERROR
The *tid* parameter is an invalid task ID.

S_taskLib_ILLEGAL_OPERATION
The operation attempted to suspend a task in another RTP.

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.

RETURNS
OK, or **ERROR** if unsuccessful.

ERRNO
S_objLib_OBJ_INVALID_ARGUMENT
name is **NULL**. *name* buffer is not valid in memory address; Or valid but it does not belong to this RTP task, so access is forbidden. e.g., an RTP task's auto variables do not belong to another task in the same RTP. Or it does belong to this RTP task but can not be read due to access control.

S_objLib_OBJ_NOT_FOUND
No task with *name* was found.

SEE ALSO
taskLib, **taskOpen( )**, **taskClose( )**

---

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 by other tasks in the same RTP. Tasks that attempt to delete a protected task block until the task is unsafe. When a task becomes unsafe, the deleter is 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.

RETURNS
OK, or **ERROR** if unable to disable task deletion safety.

ERRNO
S_taskLib_NO_TCB
The current task was created using a direct call to the **_taskOpen( )** system call and cannot be made unsafe by **taskUnsafe( )**.

SEE ALSO
taskLib, **taskSafe( )**

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( )**.

RETURNS

The most recent **tickSet( )**/**tick64Set( )** value, plus all **tickAnnounce( )** calls since.

RETURNS

current tick value

ERRNO

N/A

SEE ALSO

tickLib, **tickGet( )**, **tick64Set( )**, **tickSet( )**, **tickAnnounce( )**

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, and incremented every system clock tick.

RETURNS

The current value of the tick counter.

ERRNO

N/A

SEE ALSO

tickLib, the VxWorks programmer guides.

time( )

NAME

time( ) – determine the current calendar time (ANSI)

SYNOPSIS

```
time_t time
```

```
    (
    time_t *timer  /* calendar time in seconds */
    )
```

DESCRIPTION This routine returns the implementation's best approximation of current calendar time in seconds. If *timer* is non-**NULL**, the return value is also copied to the location *timer* points to.

RETURNS The current calendar time in seconds, or **ERROR** (-1) if the calendar time is not available.

ERRNO Not Available

SEE ALSO **time**, **clock_gettime( )**

---

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.

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

ERRNO **EINVAL**
The *timerid* specified is invalid.

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.

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.

SEE ALSO

timerLib, **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 *routine* is called in the context of the task which created the timer. This routine should be called with the timer disarmed.

The signal handling routine should be declared as:

```
    void my_handler
        (
        timer_t timerid,      /* expired timer ID */
```

2

```
        int arg                /* user argument    */
        )
```

NOTE Non-POSIX.

RETURNS 0 (**OK**), or -1 (**ERROR**) if the timer is invalid or timer is armed

ERRNO **EINVAL**
The *timerid* specified is invalid or the timer is armed.

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 * _Restrict evp,       /* user event handler */
    timer_t * _Restrict       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 process or task when the timer expires. If *evp* is **NULL**, a default signal (SIGALRM) is queued to the process, and the signal data is set to the timer ID. Initially, the timer is disarmed. The various types of asynchronous notifications that can be specified in *evp* and used when a timer expires are the following:

SIGEV_NONE
no notification occurs.

SIGEV_SIGNAL
The signal specified in sigev_signo shall be generated for the process which created the timer.

SIGEV_TASK_SIGNAL
The signal specified in sigev_signo shall be generated for the task which created the timer.

SIGEV_THREAD
A POSIX thread shall be spawned in the calling process with the attributes specified in sigev_notify_attributes and the entry point specified in sigev_notify_function. The value in sigev_value will be passed as an argument to this routine.

The timers based on the **CLOCK_REALTIME**, **CLOCK_MONOTONIC**, and thread CPU-time clocks are supported. However, only the owner of the thread CPU-time clock can create timers based on the CPU-time clock. The **CLOCK_THREAD_CPUTIME_ID** refers to the calling thread's CPU-time clock.

Note that when **SIGEV_THREAD** is specified the detach state specified in sigev_notify_attributes must be **PTHREAD_CREATE_DETACHED**.

RETURNS 0 (**OK**), or -1 (**ERROR**) if too many timers already are allocated, the signal number is invalid, or if the thread attributes specified for an event with notification type **SIGEV_THREAD** are not valid.

ERRNO

EINVAL
The specified clock ID is invalid or the signal number is not valid.

EAGAIN
The system lacks resources to handle the request.

SEE ALSO **timerLib**, **timer_delete( )**, **pthreadLib**

timer_ctl( )

NAME **timer_ctl( )** – performs a control operation on a kernel timer (system call)

SYNOPSIS

```
STATUS timerCtl
   (
   TIMER_CTL_CMD cmdCode,
   int timerId,
   void * pArgs,
   int argSize
   )
```

DESCRIPTION This routine performs the following operations on a opened/created timer. The following is the description of the *cmdCode*.

TIMER_CTL_GETTIME
get the remaining time before expiration and the reload value

TIMER_CTL_SETTIME
set the time until the next expiration and arm timer

TIMER_CTL_GETOVERRUN
return the timer expiration overrun

TIMER_CTL_MODIFY
modify the asynchronous notification mechanisim for the timer

RETURNS **OK** or number of timer exporation overrun on successfull operation. Otherwise **ERROR**.

ERRNO **EINVAL**
The name is not specified or the *timerid* specified is not valid.

EAGAIN
There is not enough resources to handle the request.

ENOSYS
The component **INCLUDE_POSIX_TIMERS** has not been configured into the kernel.

SEE ALSO **timerLib**

timer_delete( )

NAME **timer_delete( )** – remove a previously created timer (POSIX)

SYNOPSIS

```
int timer_delete
    (
    timer_t timerid  /* timer ID */
    )
```

DESCRIPTION This routine removes a timer, *timerid*, that was previously created using **timer_create( )**.

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

ERRNO **EINVAL**
The specified *timerid* is invalid.

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
The specified *timerid* is invalid.

ENOSYS
This system has not been configured with **INCLUDE_POSIX_TIMERS** to support this routine.

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
The specified *timerid* is invalid.

SEE ALSO

timerLib

timer_open( )

NAME

timer_open( ) – open a timer

SYNOPSIS

```
timer_t timer_open
    (
    const char *    name,
    int             mode,       /* OM_CREATE, ... */
    clockid_t       clock_id,   /* 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 process or task when the timer expires. If *evp* is **NULL**, a default signal (SIGALRM) is queued to the process, and the signal data is set to the timer ID. Initially, the timer is disarmed.

NOTE

This is a Non-POSIX API.

RETURNS

timer ID on success. Otherwise **ERROR**.

ERRNO

EINVAL
The name is not specified or the *clock_id* specified is not valid.

EAGAIN
There is not enough resources to handle the request.

ENOSYS
The component **INCLUDE_POSIX_TIMERS** has not been configured into the kernel.

SEE ALSO

timerLib

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 * _Restrict value,    /* time to be set */
    struct itimerspec       * _Restrict ovalue    /* previous 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.

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
The specified *timerid* is not valid.

SEE ALSO

timerLib, **timer_gettime( )**

---

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.

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.

SEE ALSO **timerLib**, **timer_open( )**, **timer_close( )**

tlsKeyCreate( )

NAME **tlsKeyCreate( )** – create a key for the TLS data

SYNOPSIS

```
TLS_KEY tlsKeyCreate (void)
```

DESCRIPTION This routine has been deprecated and will be removed in a future release. Please see **tlsLib** for more information.

This routine allocates a key for the data in the TLS. Each key is global to an RTP. Every task within the RTP has the same key for accessing the same slot within its TLS.

Key 0 or slot 0 of the TLS array is reserved for error conditions.

SMP CONSIDERATIONS

This API is not supported in SMP mode. Calling this routine will error and by default, will terminate the RTP.

RETURNS key for TLS data requested, or **ERROR**.

ERRNOS Possible errno values:

S_tlsLib_MAX_KEYS
Maximum number of keys has been encountered.

SEE ALSO **tlsOldLib**

tlsSizeGet( )

NAME **tlsSizeGet( )** – Get size of the TLS structure

SYNOPSIS

```
int tlsSizeGet (void)
```

DESCRIPTION This routine has been deprecated and will be removed in a future release.

This routine returns the size of the task local storage for the current RTP.

SMP CONSIDERATIONS

This API is not supported in SMP mode. Calling this routine will have undefined results in SMP.

RETURNS size of TLS for the current RTP

ERRNOS N/A

SEE ALSO **tlsOldLib**

tlsValueGet( )

NAME **tlsValueGet( )** – get a value of a specific TLS data

SYNOPSIS

```
void * tlsValueGet
    (
    TLS_KEY key
    )
```

DESCRIPTION This routine has been deprecated and will be removed in a future release.

This routine get the TLS data value associated with the specified *key*. The TLS data returned is for the current task.

SMP CONSIDERATIONS

This API is not supported in SMP mode. Calling this routine will have undefined results in SMP.

RETURNS value of TLS data, or **NULL**.

ERRNOS Possible errnos are:

S_tlsLib_INVALID_KEY
The key provided is an invalid key that is out of range or is 0.

S_tlsLib_NO_TLS
The task has no TLS area.

SEE ALSO **tlsOldLib**

tlsValueSet( )

NAME **tlsValueSet( )** – set the value of a TLS data

SYNOPSIS

```
STATUS tlsValueSet
    (
    TLS_KEY key,
    void *  value
    )
```

DESCRIPTION This routine has been deprecated and will be removed in a future release.

This routine sets the *value* in the TLS associated with the given *key*. The value set is for the current executing task. If the key is invalid, **ERROR** is returned and the value is not set for the task.

SMP CONSIDERATIONS

This API is not supported in SMP mode. Calling this routine will have undefined results in SMP.

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

ERRNOS Possible errnos are:

S_tlsLib_INVALID_KEY
The key provided is an invalid key that is out of range or is 0.

S_tlsLib_NO_TLS
The task has no TLS area.

SEE ALSO **tlsOldLib**

uname( )

NAME **uname( )** – get identification information about the system

SYNOPSIS

```
int uname
    (
    struct utsname * pName  /* where to store identification information */
    )
```

DESCRIPTION This routine provides identification information about the system. It stores them in the structure pointed to by *pName*.

Identification information are as follows:

sysname
: holds the name "VxWorks".

nodename
: holds the network name of the system as reported by **gethostname( )**.

release
: the implementation release level is mapped on VxWorks's full version number (major.minor.maintenance). This field may also hold addditional data, such as "SMP", when applicable.

version
: the version information is currently reserved for future use and simply reports "reserved".

machine
: holds the model of the BSP on which the system is running as reported by **sysModel( )**.

endian
: the architecture's endianness, big or little, as set by the BSP.

kernelversion
: the VxWorks kernel version as reported by **kernelVersion( )**.

processor
: holds the CPU family.

bsprevision
: the BSP revision level as reported by **sysBspRev( )**.

builddate
: the OS build date.

Unavailable information will be indicated by the "unknown" string.

RETURNS 0 to indicate success or -1 otherwise.

ERRNO

EINVAL
when the value of the *pName* argument is not valid.

SEE ALSO

uname, **gethostname( )**, **sysModel( )**, **sysBspRev( )**

unlink( )

NAME

unlink( ) – unlink a file

SYNOPSIS

```
int unlink
   (
    const char *name   /* path name of the file to unlink */
   )
```

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( )**

unsetenv( )

NAME

unsetenv( ) – remove an environment variable (POSIX)

SYNOPSIS

```
int unsetenv
    (
    const char * envVarName  /* name of environment variable to remove */
    )
```

DESCRIPTION

This routine removes an environment variable *envVarName* from the global environment if the variable already exists. If the variable *envVarName* does not exist the existing environment is not modified.

The variable name may not be **NULL**, the empty string or hold a "=" character.

RETURNS

0 for success, **EINVAL** if the variable name is not valid.

ERRNO

N/A

SEE ALSO **setenv**, **setenv( )**, **getenv( )**

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( )**

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 RTP heap

SYNOPSIS
```
void * valloc
    (
    unsigned size  /* number of bytes to allocate */
    )
```

DESCRIPTION
This routine allocates a buffer of *size* bytes from the RTP heap partition. 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
ENOMEM / S_memLib_NOT_ENOUGH_MEMORY
There is no free block large enough to satisfy the allocation request.

SEE ALSO
memLib

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.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

ERRNO Not Available

SEE ALSO **fioLib**, **fdprintf( )**

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.

RETURNS The number of characters output, not including the **NULL** terminator.

ERRNO Not Available

SEE ALSO **fioLib**, oprintf, **printf( )**

vxAtomicAdd( )

NAME **vxAtomicAdd( )** – atomically add a value to a memory location

SYNOPSIS

```
atomicVal_t vxAtomicAdd
    (
    atomic_t *  target,
    atomicVal_t value
    )
```

DESCRIPTION This routine atomically adds **target* and *value*, placing the result in **target*. The operation is done using signed integer arithmetic.

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,
    atomicVal_t value
    )
```

DESCRIPTION This routine atomically performs a bitwise AND operation of **target* and *value*, placing the result in **target*.

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
    )
```

DESCRIPTION This routine atomically clears **target* and returns the old value that was in **target*. This routine is intended for software that needs to atomically fetch and clear the value of a memory location.

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
    )
```

DESCRIPTION This routine atomically decrements the value in **target*. The operation is done using unsigned integer arithmetic.

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
    )
```

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.

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
    )
```

DESCRIPTION This routine atomically increments the value in **target*. The operation is done using unsigned integer arithmetic.

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,
    atomicVal_t value
    )
```

DESCRIPTION This routine atomically performs a bitwise NAND operation of **target* and *value*, placing the result in **target*.

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,
    atomicVal_t value
    )
```

DESCRIPTION

This routine atomically sets **target* to *value* and returns the old value that was in **target*. This routine is intended for software that needs to atomically fetch and replace the value of a memory location.

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,
    atomicVal_t value
    )
```

DESCRIPTION

This routine atomically subtracts *value* from **target*, placing the result in **target*. The operation is done using signed integer arithmetic.

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,
    atomicVal_t value
    )
```

DESCRIPTION

This routine atomically performs a bitwise XOR operation of **target* and *value*, placing the result in **target*.

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,
    atomicVal_t oldValue,
    atomicVal_t newValue
    )
```

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*.

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.

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);
}
```

RETURNS A set of CPUs that have been enabled.

ERRNO N/A

SEE ALSO **vxCpuLib**, **vxCpuConfiguredGet( )**, cpuset

wait( )

NAME **wait( )** – wait for any child RTP to terminate (POSIX)

SYNOPSIS

```
int wait
    (
    int            * pStatus  /* return status */
    )
```

DESCRIPTION
This routine suspends the calling task until a child RTP terminates.

RETURNS
child's RTP ID if woken up by child signal, or -1 if no child processes waiting, or the call was interrupted by a signal.

ERRNO
ECHILD
The calling process has no existing unwaited-for child processes.

EINTR
The function was interrupted by a signal. The value of the location pointed to by pStatus is undefined.

SEE ALSO
sigLib

---

waitpid( )

NAME
waitpid( ) – Wait for a child process to exit, and return child exit status

SYNOPSIS
```
pid_t waitpid
    (
   pid_t childRtpId,
   int * pStatus,
   int   options
    )
```

DESCRIPTION
This routine suspends the calling task until delivery of a signal. If the signal that occurs is SIGCHLD, the child's process ID and exit status are returned. The options parameter is a bit mask where the following values are supported -

WNOHANG
If no processes wish to report status, 0 is returned.

WUNTRACED
Children of the current process that are stopped due to a **SIGSTOP** signal also have their status reported.

RETURNS
-1 or child's PID if woken up by child signal

ERRNO
EINTR
EINVAL.

SEE ALSO
rtpLib, the VxWorks programmer guides

write( )

NAME

write( ) – write bytes to a file

SYNOPSIS

```
ssize_t write
   (
   int fd,
   const void * buffer,
   size_t nbytes
   )
```

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

wvEvent( )

NAME

wvEvent( ) – record a System Viewer user event

SYNOPSIS

```
void wvEvent
   (
   event_t    usrEventId,      /* event */
   char *     buffer,          /* buffer */
   size_t     bufSize          /* buffer size */
   )
```

2

DESCRIPTION none

RETURNS **OK**, or **ERROR** if the event can not be logged.

ERRNO N/A

SEE ALSO **wvScLib**

---

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.

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.